Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2010,7
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Deductive Verification of a Byzantine Agreement Protocol

Roman Krenický, Mattias Ulbrich

2010

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fakultät für Informatik

Please note:
This Report has been published on the Internet under the following Creative Commons License:
http://creativecommons.org/licenses/by-nc-nd/3.0/de.

Deductive Verification of a Byzantine Agreement Protocol

Roman Krenický and Mattias Ulbrich

Karlsruhe Institute of Technology
Institute for Theoretical Computer Science
D-76128 Karlsruhe, Germany
mattias.ulbrich@kit.edu

April 20, 2010

Abstract

This report describes a formalisation and deductive verification of a Byzantine Agreement Protocol. The model evolves over twelve steps of refinement each introducing a new aspect. The Event-B method is used to model the protocol, and the publicly available tool Rodin is used to deductively prove its correctness.

Contents

1 Introduction **2**
- 1.1 Problem statement . . . 2
- 1.2 Assumptions . . . 2
- 1.3 Rounds . . . 3

2 Modelling the Byzantine Agreement Problem **4**
- 2.1 The contexts of the model . . . 4
- 2.2 The hierarchy of machines . . . 5
- 2.3 Machine "MESSAGES" . . . 6
- 2.4 Machine "MESSAGESSIGNED" . . . 6
- 2.5 Machine "HISTORY" . . . 8
- 2.6 Machine "GUARANTEES" . . . 9
- 2.7 Machine "HYBRIDGUARANTEES" . . . 12
- 2.8 Machine "VALUETABLES" . . . 13
- 2.9 Machine "ZA" . . . 15
- 2.10 Machine "ROUNDLESS" . . . 17
- 2.11 Machine "SM" . . . 19

3 Important Properties **21**
- 3.1 Selected lemmata of VALUETABLES . . . 23
- 3.2 Sketch for the proof of `agreement_subset` . . . 24
- 3.3 On the invariant `agreement_voting` . . . 25

4 Byzantine Agreement Protocols **26**

5 Definitorial Extensions **27**

6 Lessons Learnt **28**
- 6.1 Event-B as modelling language . . . 28
- 6.2 The tool Rodin . . . 28

7 Conclusion **29**

1 Introduction

Byzantine Agreement Protocols are used to ensure that in the presence of a limited number of defect or malicious units a message can be distributed amongst them in such a way that, eventually, all correctly working units agree on one consensus decision. They can be used, for instance, to synchronise the view components of a system have about each other. The original problem statement and the first protocols were presented in [9] and [5]. Many protocols that mainly differ in the applied authentication scheme or the fault model have been presented since. See Sect. 4 for a brief overview over a number of protocols.

Agreement protocols are of relevance today when in a safety-critical environments several components of a decentralised system need to share a common value. If no central authority instance is present, the components have to perform an agreement protocol to come up with a consentaneous decision.

Byzantine agreement protocols are not too complex in their nature and can be described concisely. They are, on the other hand, also not trivial algorithms, and Lamport et al. admitted in [5]: "We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm." They are therefore most appropriate for a formal examination. A protocol variation without secure signatures (called *oral messages*) has been formally verified in [7] using the higher order proof environment PVS. We will here concentrate on a different kind of protocol (*written* or *signed messages*) which uses secure signatures.

The formalism chosen for our formalisation is Event-B [1] and the tool used for the verification is Rodin[1] [2].

1.1 Problem statement

We consider a system which is composed of a non-empty set of MODULEs. Every module is either **non-faulty**, **arbitrarily faulty** or **symmetrically faulty**. The union of the (disjoint) sets of arbitrarily faulty and symmetrically faulty modules is the set of **faulty** modules.

Modules receive and send messages which contain VALUEs. A non-faulty module relays every message it receives to all other modules which have not yet seen this message. Faulty modules do the same, but they may **drop** messages instead of passing them on. Arbitrarily faulty modules are unrestricted in the messages that they discard. A symmetrically faulty module either relays a message to *all* modules which have not seen it yet (like a non-faulty module) or it does not send *any* relayed message at all. Their behaviour may change, however, from received message to message. Please note: Faulty modules can never *forge* messages, only discard them.

The protocol is divided into steps, called rounds. In the first round (**round 0**), a dedicated module **transmitter** sends messages to all other modules. A non-faulty transmitter sends the intended value V_0 to all other modules. A faulty transmitter, however, may send arbitrary initial messages to all parties, it may also drop messages, i.e., not send a message to every other module.

From round 1 on, the transmitter does no longer receive or send any more messages. Messages are amended with a history of those nodes which have already seen it. A receiving node may therefore easily detect to whom a message has to be passed on and to whom not.

In the original protocol an **error value E** was used to describe faulty behaviour. We opted for "dropping" messages, but one formulation can easily simulate the other.

1.2 Assumptions

We list here the assumptions that we make about the modules involved in the protocol:

A1 *The sending of messages is synchronised.*

Equivalent: *The protocol is strictly round-based.*

No module can receive a message of a certain round while it still can produce messages of the same round. Messages belong to a certain round and cannot be delivered at a later round.

A2 *Sender and receiver references embedded in a message are truthful.*

[1] We first used Rodin release 1.0 and later 1.1.

We consider this integral, reliable information part of the network environment or given due to the communication wiring.

A3 *The transmitter sends at most one message to each module in the first round. It does not send any more messages in later rounds.*

This assumptions can easily be implemented in reality: If a module encounters more than one different message stemming from the transmitter, it can discard all of them (assuming the sender must be faulty) or all but the first for instance.

A4 *Messages cannot be forged.*

A module (even a faulty one) cannot make up a message from scratch. A module must receive a message of the previous round to produce one (or more) messages for the upcoming round. All it can do is amend its own signature to extend the set of visited modules.

It is up to the used signature and message scheme to ensure that this is indeed the case. Usually this means that forged messages can be created but are invalid (e.g. digital signatures violated) and, hence, forgery can always be detected. The assumption requires also that messages must be sent immediately, i.e. in the round directly after receiving the message.

A5 *Messages carry histories.*

Every message contains, besides its value, the set of modules through which it has passed. We assume that A4 also covers the histories of messages.

A6 *There is a response time within which every non-arbitrarily-faulty module is guaranteed to relay a message (if it relays it).*

An arbitrarily malfunctioning module may here delay a message for an arbitrary period of time. This assumption is needed for an asynchronous version of the protocol in which A1 does not hold and timeout deadlines are employed.

We will in later stages of the refinement chain relax A1 (see Sect. 2.10).

1.3 Rounds

The perception of when a round begins or ends varies throughout the literature. This section clarifies our notion on what happens during a round and how rounds are counted.

In our model, messages which are being sent (are in *rec*) in round n arrive in round $n+1$ (Fig. 1). Therefore, in round n (for $round = n$) n sending rounds are finished, i. e. the respective messages have arrived. In the original description of ZA [3] on the other hand, messages which are being sent in round n arrive in the same round (Fig. 2). Accordingly, in round n (effects of ZA(n)) $n+1$ sending rounds are finished, i. e. the respective messages have arrived. Hence, the effects of ZA(n) can be observed in the state with $round = n + 1$ in our model, because in both cases $n + 1$ sending rounds are finished.

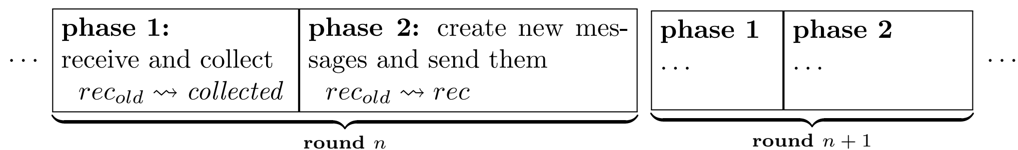


Figure 1: Rounds of our model — $round = n$: n sending rounds finished, i. e. messages have arrived; rec_{old} refers to the contents of *rec* in the previous round

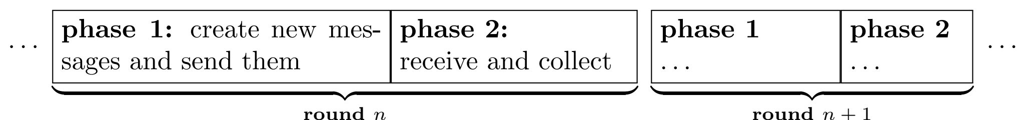


Figure 2: Rounds of ZA — $r = n$ [ZA(n)]: $n+1$ sending rounds finished, i. e. messages have arrived

Summing up, for $round = n$ exactly n sending rounds have been finished (i. e. the respective messages have arrived and been observed, that is entered in *collected*) and the $n + 1^{st}$ sending round is in progress (the messages are on their way).

In [3] the bound for ZA is given as $r \geq \text{card}(\mathit{faulty})$. This is due to the difference in counting the rounds (see Sect. 1.3). Both, for our $round \geq \text{card}(\mathit{faulty}) + 1$ and for $\text{ZA}(r)$ with $r \geq \text{card}(\mathit{faulty})$ there are $\text{card}(\mathit{faulty}) + 1$ sending rounds finished. Accordingly, the bounds are identical (with respect to the number of sending rounds).

2 Modelling the Byzantine Agreement Problem

Event-B is a formal method for modelling discrete reactive systems. It is based on the concept of *refinement* allowing to formally derive more detailed system descriptions from more abstract ones. Even though the models can be used for model checking [6] and model animation [8], they are most often verified using deductive reasoning: Formal proofs have to be given to show the correctness of refinements and invariants. For an introduction and overview over the Event-B method, see [4], for instance.

The concept of dividing a model into several levels of refinement allows us to distribute different aspects of the modelling task among several steps of the design. For one machine, the specifier, the verifier and an interested reader can then concentrate on one particular aspect of the model instead of dealing with the entire complexity all the time. We employ the means of refinement in our model beginning with very simple models which implement the postulated assumptions, and then introduce more complex elements of the protocol in later steps. The descriptions and their proven properties become more and more complex.

Models in Event-B are defined in contexts (where carrier sets, constants and axioms are provided) and in machines (which contain definitions of events and invariants which descripe the discrete state transition system). We will in this section first present the contexts used by the various machines and then—in a top-down fashion—describe the refinement tower of Event-B machines.

Notational remarks

We will use the following fonts to distinguish different kinds of identifiers:

MachineName	Name of a context or machine
`CarrierName`	Name of a carrier set
constName	Name of a constants or variable
`act1`	Name of a named element (guard, action, invariant, . . .)
`thm1`	Name of a theorem (which is implied by before standing invariants or axioms)

We declare events *extended* if they have substitutions in common with the event they refine. In parentheses, we list the actions which are "inherited" from the refined event, i.e., which are implicitly copied verbatim.

2.1 The contexts of the model

The contexts used throughout the model are depicted in Fig. 3, the more complex context ModuleList is covered in Sect. 2.9.

The sets of `Module`s and messages (`Value`s) are modelled as carrier sets in the first context Context to emphasise their role as primary objects in the protocol. Please note that it is not necessary to request `Module` to be a finite set. All proofs work also for an infinite number of modules. However, the subset of *faulty* modules is chosen to be a finite set. We later want to argue that the number of rounds the protocol has to perform is related to the number of faulty nodes—which is the cardinality of this set—and, hence, need to know that this set is always finite. We call the `Value` that is the value which ought to be broadcast the *intended value* and denote it as V_0. The *transmitter* is an arbitrary `Module` acting as the sole sender in round 0. If the transmitter is not arbitrarily faulty, V_0 is the only value observed during the algorithm.

The context HybridContext refines the notion of a faulty module by introducing two sets *arbFaulty* and *symFaulty* which partition the set of *faulty* modules, i.e., the sets are disjoint and their union is *faulty*. The context VotingContext is used in a later refinement (ValueTable, see

Sect. 2.8) where an arbitrary voting function is considered. A voting function is a total function from value tables (i.e. partial functions from modules to values) to values. The only requirement is that if the table is not empty, the chosen value must be one of the values in the range of the table (axm_vote1).

(a) the abstract context Context

(b) the more detailed context HybridContext adds to Context

(c) the context introducing the notion of a voting function

Figure 3: Contexts used by the machines in the model

2.2 The hierarchy of machines

We have structured our model into twelve machines. Five of them (namely Messages, MessagesSigned, History, Guarantees, HybridGuarantees) gradually introduce new aspects into the model in a linear fashion. Two (namely ValueTables, ZA) extend the chain to model the protocol variant ZA (e.g. described in [3]). Another two (namely Roundless and SM) weaken the concept of the round-based protocols and finally model the standard-algorithm SM.

The remaining machines (with their names ending in Tech) are definitorial extensions (see Sect. 5) of their respective counterparts and are used to add variables helpful for the agreement proofs. To keep the modelling free of too many technical details, we introduced the technical machines, in which we proof the agreement properties. These invariants are then invariants also in the original machines by the theorem from Sect. 5. The figures presenting the machines in the following do not always contain all proven invariants but concentrate on essential properties to keep the presentation relatively clear. Events, however, are depicted complete.

Please see Fig. 4 for a schematic of the refinement hierarchy. Therein, contexts are drawn as

rhomboids, machines as rectangles. A solid line indicates a refinement relationship (going from refin*ing* to refin*ed* entity), and a dotted line stands for a "`sees`"-clause between a machine and a context.

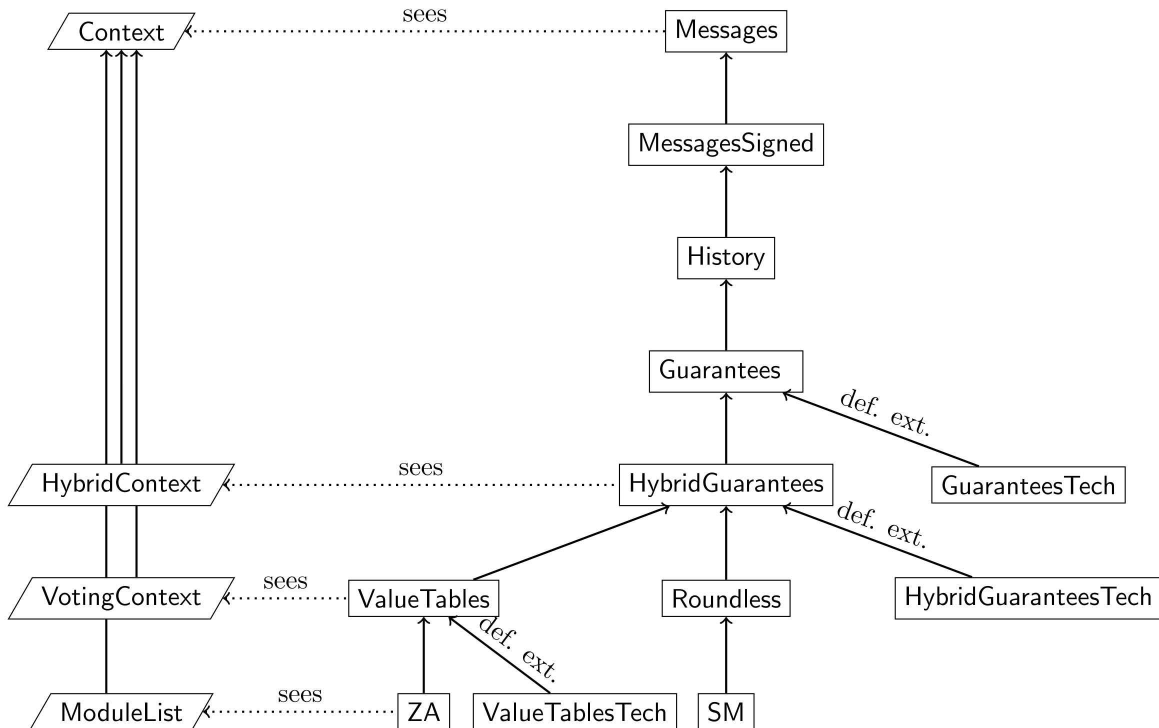


Figure 4: Refinement Hierarchy of Machines

2.3 Machine "Messages"

The initial machine MESSAGES—depicted in Fig. 5—introduces the general concepts of messages, rounds and the collection of values. For most of the upcoming machines (all but ROUNDLESS and SM), the protocol is strictly round based by assumption A1, and we memorise the current round number in a variable *round*. For a description of the notion of a round please see Sect. 1.3. The event structure with the two events Initialisation and Round is kept up throughout this model.

A single message is an element of the Cartesian product `MODULE` $\times$ `MODULE` $\times$ `VALUE` with the first component the sender module of the message, the second its receiver and the third the value transmitted by it. The variable *messages* holds the set of messages sent in the current round (i.e. in round *round*). Every module keeps an account of the values which it has already received. We model this by a total function *collected* which maps to every `MODULE` a set of `VALUE`s.

Action `act1` of the initialisation is given in form of a before-after-predicate because the result value *messages'* is used in the initial assignment to *collected*. This shape of a non-deterministic "such-that" substitution will remain the same in the upcoming refinements. An arbitrary (partial) mapping between `MODULE`s and `VALUE`s is used to describe the initial set of messages (which the transmitter sends to modules) and the initial mapping of collected values. The first action of event Round imposes hardly any restriction on a refining event. Almost every set of messages can be used to fulfill this indeterministic choice. `act2` which updates the sets of encountered values by those values sent in the last round, has to be adapted with the representation of the sent messages. Action `act3` which initialises and increments the round counter remains the same in all future machines.

Please note that `act1` in the initialisation and round together ensure—already at this high level of abstraction—assumption A3 by explicitly allowing/disallowing the *transmitter* as sender of messages.

2.4 Machine "MessagesSigned"

In a first refinement, we introduce a restriction on the set of messages which can be sent by the modules in the model. According to assumption A4, modules can only pass on values they receive

MACHINE MESSAGES
SEES CONTEXT
VARIABLES

messages
round
collected

INVARIANTS

type_messages : $messages \in (\mathrm{MODULE} \times \mathrm{MODULE}) \leftrightarrow \mathrm{VALUE}$
type_round : $round \in \mathbb{N}$
type_collected : $collected \in \mathrm{MODULE} \rightarrow \mathbb{P}(\mathrm{VALUE})$

EVENTS

Initialisation

begin

act1 : $messages, collected :|$
$messages' = \{transmitter\} \times \mathrm{MODULE} \times \mathrm{VALUE} \land$
$collected' = (\mathrm{MODULE} \times \{\varnothing\}) \mathbin{\lhd\!\!\!+} \{transmitter \mapsto ran(messages')\}$
act3 : $round := 0$

end

Event *ROUND* $\widehat{=}$

begin

act1 : $messages :\in ((\mathrm{MODULE} \setminus \{transmitter\}) \times \mathrm{MODULE}) \leftrightarrow \mathrm{VALUE}$
act2 : $collected := \lambda n \cdot n \in \mathrm{MODULE} \mid collected(n) \cup \{s, v \cdot (s \mapsto n) \mapsto v \in messages \mid v\}$
act3 : $round := round + 1$

end

END

Figure 5: Machine MESSAGES

and never come up with new values which were not presented to them in a message of the previous round.

The refined events change very little, but `act1` of event ROUND is modified to only allow messages which are in accordance with A4. Only messages which are induced by a message of the current round may be used in the next round. The information about sender and receiver in a message tuple is authentic and cannot be faked as we may assume by A2.

Already at this high abstraction level we can observe two rather important invariants which will contribute a lot to the notion of validity. The invariant

$$\forall s, r, v \cdot (s \mapsto r) \mapsto v \in \mathit{messages} \Rightarrow v \in \mathit{collected}(\mathit{transmitter})$$

(named `no_new_vals`) captures the fact that every value appearing in a message at any time must have appeared in the first round at the transmitter already. Invariant `no_new_vals2`

$$\forall n \cdot \mathit{collected}(n) \subseteq \mathit{collected}(\mathit{transmitter})$$

states the corresponding property for the values collected by modules: They must have appeared in the collection of the transmitter first.

MACHINE MessagesSigned
REFINES Messages
SEES Context
VARIABLES
  messages
  round
  collected
INVARIANTS
  `no_new_vals :` $\forall s, r, v \cdot (s \mapsto r) \mapsto v \in \mathit{messages} \Rightarrow v \in \mathit{collected}(\mathit{transmitter})$
  `no_new_vals2 :` $\forall n \cdot \mathit{collected}(n) \subseteq \mathit{collected}(\mathit{transmitter})$
EVENTS
Initialisation
  (inherits: `act1`, `act3`)
  begin
  end
Event *ROUND* $\widehat{=}$
refines *ROUND (inherits: `act2`, `act3`)*
  begin
    `act1 :`
      $\mathit{messages} :\in \mathbb{P}(\{s, r, v, n \cdot (s \mapsto r) \mapsto v \in \mathit{messages} \wedge r \neq \mathit{transmitter} \mid (r \mapsto n) \mapsto v\})$
  end
END

Figure 6: Machine MessagesSigned

2.5 Machine "History"

The second refinement to machine History (cf. Fig. 7) implements A5 hereby implying a change of representation for the set of messages. Instead of the state variable *messages*, we will now employ a variable *rec*[2] which holds a set of messages with histories. We need not concern ourselves with the order of modules in histories yet, but model them as *sets* of modules. A message is, hence, an element

[2]standing for "received messages"

of the product $(\texttt{MODULE} \times \texttt{MODULE}) \times (\mathbb{P}(\texttt{MODULE}) \times \texttt{VALUE})$. At the same time we ensure that no message is ever sent to a module which has already "seen" it.

This representation change gives rise to a glueing invariant `glue_rec_messages`

$$messages = \{s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \mid (s \mapsto r) \mapsto v\}$$

stating that the tuples in *messages* is obtained from the set of messages tuples with history by dropping with the history component. Please note that more than one message in *rec* can fall onto the same message in *messages* if they coincide in every component but the history list.

Histories are not arbitrary sets, they have some fundamental properties by construction, of which the most important are captured in `rec_content`

$$\forall s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \Rightarrow s \in l \land transmitter \in l \land r \notin l$$

stating that the sender s of a message and the *transmitter* are always part of the history l, while, on the other hand, the receiver r never is. We change the nature of histories in the refinement to ZA where we take the order into consideration, please see Sect. 2.9 for details.

In the beginning (round 0) every history is set to $\{transmitter\}$ and it grows by exactly one module per round which manifests itself in invariant `set_card`

$$\forall s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \Rightarrow \text{finite}(l) \land \text{card}(l) = round + 1 \ .$$

The event description undergoes a more thorough change than in previous steps due to the representation change, but captures the same essential ideas. The events `act1` in both events ensure that histories are used the way they ought to be: In round 0 they are initialised with the singleton set $\{transmitter\}$ and in later rounds the new sender r of any outgoing message is added to the history list. Since the variable *message* was originally changed by a generalised substitution with before-after-predicate, we have to provide a witness (in the `with` clauses) which fulfills the requirements given in the refined machine and fits the needs of this refinement.

2.6 Machine "Guarantees"

It is only now (see Fig. 8) that we introduce the protocol-specific message handling. In machine GUARANTEES we distinguish between faulty and non-faulty modules. We assume that non-faulty nodes behave as the protocol requires while faulty may behave arbitrarily.

In event Round we use an expression of the form "let $R = \alpha$ in $\beta(R)$" which is not syntactically valid in Event-B, but was introduced for better readability. This let-in-expression can be equivalently written as $\forall R \cdot R = \alpha \Rightarrow \beta(R)$. The locally bound identifier R holds the set of messages which would be sent if all modules were non-faulty, and is obviously an upper bound for the set of the messages actually sent ($rec' \subseteq R$). We also require that for any non-faulty sender all messages are properly relayed, i.e. that any message is forwarded to all module which have not yet received it.

This is the level of refinement at which we reason about an agreement property for the first time. Since the non-faulty modules' processing is total, distribution of values amongst all non-faulty modules can be guaranteed. The invariant `agreement`[3]

$$round \geq \text{card}(faulty) + 1 \Rightarrow (\forall n, m \cdot n \notin faulty \land m \notin faulty \Rightarrow collected(n) = collected(m))$$

states that after a certain number of rounds ($rounds \geq \text{card}(faulty) + 1$), the observations of all non-faulty modules coincide ($collected(n) = collected(m)$). Please see Sect. 3 on a more detailed analysis of this property. Property `ex_non_faulty`

$$round \geq \text{card}(faulty) \Rightarrow (\forall s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \Rightarrow (\exists x \cdot x \in l \land x \notin faulty))$$

is in the very center of the agreement arguments: If the round counter is equal to or greater than the number of faulty modules, then there must be at least one module which is non-faulty (simple cardinality considerations) in the history of every message which has seen and, hence, distributed the value to all other modules. This is the main argument why after $\text{card}(faulty) + 1$ rounds all non-faulty modules have seen the same set of values. We can also prove invariant `validity`

$$round \geq 1 \land transmitter \notin faulty \Rightarrow (\forall n \cdot collected(n) = \{V_0\})$$

now, stating that in case of a working transmitter after the first round every module has experienced the value V_0 and only this value.

[3]in GUARANTEESTECH

MACHINE History
REFINES MessagesSigned
SEES Context
VARIABLES

rec
$round$
$collected$

INVARIANTS

`type_rec` : $rec \in (\texttt{MODULE} \times \texttt{MODULE}) \leftrightarrow (\mathbb{P}(\texttt{MODULE}) \times \texttt{VALUE})$

`glue_rec_messages` : $messages = \{s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \mid (s \mapsto r) \mapsto v\}$

`rec_content` : $\forall s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \Rightarrow s \in l \wedge transmitter \in l \wedge r \notin l$

`no_new_vals` : $\forall s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \Rightarrow v \in collected(transmitter)$
direct consequence of MessagesSigned.`no_new_vals`

`trans_not_dom` : $\forall s \cdot s \mapsto transmitter \notin dom(rec)$
direct consequence of `rec_content`

`set_card` : $\forall s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \Rightarrow \mathrm{finite}(l) \wedge \mathrm{card}(l) = round + 1$
due to construction the length of the history depends on the round

EVENTS
Initialisation
(inherits: `act3`*)*
begin
with
messages' : $messages' = \{s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec' \mid (s \mapsto r) \mapsto v\}$
`act1` : $rec, collected :\mid$
$rec' \subseteq \{transmitter\} \times (\texttt{MODULE} \setminus \{transmitter\}) \times (\{\{transmitter\}\} \times \texttt{VALUE}) \wedge$
$collected' = (\texttt{MODULE} \times \{\varnothing\}) \lhd\!\!- \{transmitter \mapsto ran(ran(rec'))\}$
end
Event $ROUND \mathrel{\widehat{=}}$
refines $ROUND$ *(inherits:* `act3`*)*
begin
with
messages' : $messages' = \{s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec' \mid (s \mapsto r) \mapsto v\}$
`act1` : $rec :\in \mathbb{P}(\{s, r, l, v, n \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \wedge n \notin l \wedge n \neq r \mid$
$(r \mapsto n) \mapsto (l \cup \{r\} \mapsto v)\})$
`act2` : $collected := \lambda n \cdot n \in \texttt{MODULE} \mid collected(n) \cup \{s, l, v \cdot (s \mapsto n) \mapsto (l \mapsto v) \in rec \mid v\}$
end
END

Figure 7: Machine History

MACHINE GUARANTEES
REFINES HISTORY
SEES CONTEXT
VARIABLES

rec
round
collected

INVARIANTS *some omitted, some lifted from* GUARANTEESTECH

`nonfaulty_transmitter` : $transmitter \notin faulty \Rightarrow collected(transmitter) = \{V_0\}$

`ex_nonfaulty` : $round \geq \mathrm{card}(faulty) \Rightarrow (\forall s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \Rightarrow (\exists x \cdot x \in l \wedge x \notin faulty))$

`validity` : $round \geq 1 \wedge transmitter \notin faulty \Rightarrow (\forall n \cdot collected(n) = \{V_0\})$

`agreement_subset` : $round \geq \mathrm{card}(faulty) + 1 \Rightarrow (\forall n, m \cdot n \notin faulty \wedge m \notin faulty \Rightarrow collected(n) \subseteq collected(m))$
See proof sketch in Sect. 3.2 (lifted)

`agreement` : $round \geq \mathrm{card}(faulty) + 1 \Rightarrow (\forall n, m \cdot n \notin faulty \wedge m \notin faulty \Rightarrow collected(n) = collected(m))$
Main theorem, simple consequence of `agreement_subset` (lifted)

finite_running : $\mathrm{finite}(MODULE) \wedge round \geq \mathrm{card}(MODULE) \Rightarrow rec = \varnothing$
The protocol terminates for a finite number of modules (lifted)

EVENTS

Initialisation

(inherits: `act3`*)*
begin
`act1` : $rec, collected :|$
$\exists values \cdot values \in MODULE \mathrel{+\!\!\!\rightarrow} VALUE \wedge$
$(transmitter \notin faulty \Rightarrow values = MODULE \times \{V_0\}) \wedge$
$rec' = \{n \cdot n \in dom(values) \setminus \{transmitter\} \mid$
$(transmitter \mapsto n) \mapsto (\{transmitter\} \mapsto values(n))\} \wedge$
$collected' = (MODULE \times \{\varnothing\}) \mathbin{\lhd\!\!\!-} \{transmitter \mapsto ran(values)\}$
end

Event $ROUND \mathrel{\widehat{=}}$
refines *ROUND (inherits:* `act2, act3`*)*
begin
`act1` : $rec :|$
let $R = \{l, v, s, r, n \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \wedge n \neq r \wedge n \notin l \mid (r \mapsto n) \mapsto ((l \cup \{r\}) \mapsto v)\}$ in
$rec' \subseteq R \wedge$
$(\forall t, u, x, w \cdot (t \mapsto u) \mapsto (x \mapsto w) \in R \wedge t \notin faulty \wedge w \notin collected(t) \Rightarrow$
$(t \mapsto u) \mapsto (x \mapsto w) \in rec')$
end

END

Figure 8: Machine GUARANTEES

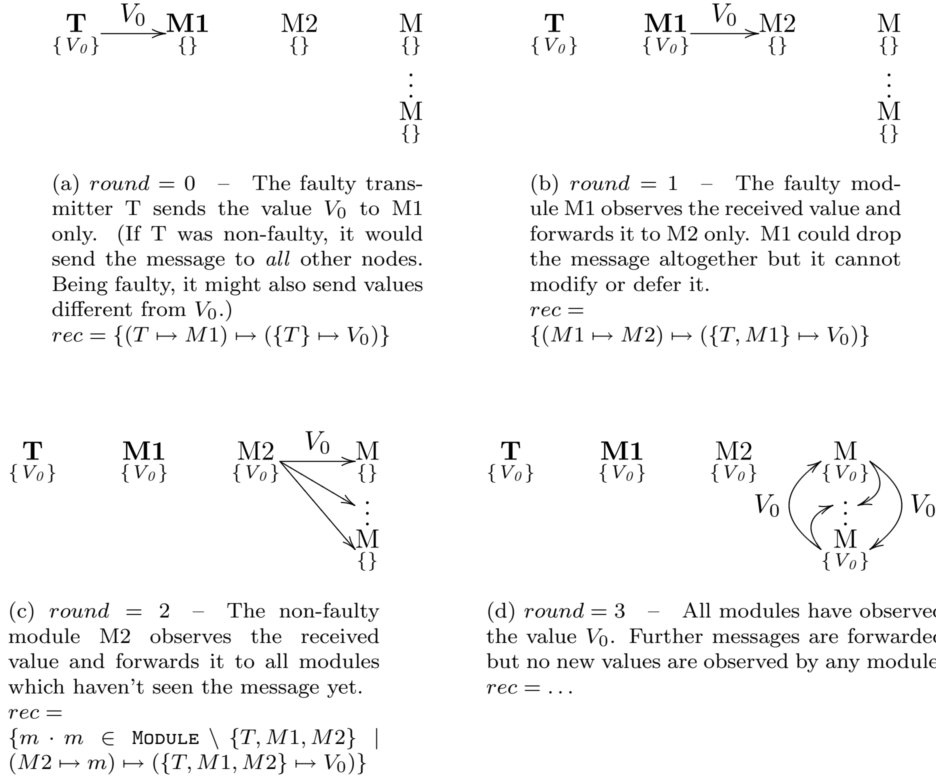
2.6.1 Lifting

For each of this machine and the following two (HybridGuarantees and ValueTables) we have introduced a technical companion (with suffix Tech) which contains technical details on the model and proofs. They are designed as definitorial extensions implying that some results can be lifted to the refined machines directly. Please see Sect. 5 for a formal explanation.

The definitorial extension has got two more variables *rec_old* and *collected_old* which store the value of *rec* and *collected* respectively of the previous round. They are needed due to the inductive nature of some lemmata proofs. Their assignment is the only change to the event descriptions. Apart from that, only additional invariants are provided.

2.6.2 Example

An example scenario with two faulty modules, one of them the transmitter, is shown in Fig. 9. In round 0 (Fig. 9(a)), the transmitter sends the initial value, which arrives and is forwarded in the following round (Fig. 9(b)). One can see, that in round 3 all modules have received the value sent by the transmitter.



(a) *round* = 0 – The faulty transmitter T sends the value V_0 to M1 only. (If T was non-faulty, it would send the message to *all* other nodes. Being faulty, it might also send values different from V_0.) $rec = \{(T \mapsto M1) \mapsto (\{T\} \mapsto V_0)\}$

(b) *round* = 1 – The faulty module M1 observes the received value and forwards it to M2 only. M1 could drop the message altogether but it cannot modify or defer it. $rec = \{(M1 \mapsto M2) \mapsto (\{T, M1\} \mapsto V_0)\}$

(c) *round* = 2 – The non-faulty module M2 observes the received value and forwards it to all modules which haven't seen the message yet. $rec = \{m \cdot m \in \texttt{Module} \setminus \{T, M1, M2\} \mid (M2 \mapsto m) \mapsto (\{T, M1, M2\} \mapsto V_0)\}$

(d) *round* = 3 – All modules have observed the value V_0. Further messages are forwarded but no new values are observed by any module. $rec = \ldots$

Figure 9: In this example, transmitter T and module M1 are faulty, the other modules are non-faulty, hence card(*faulty*) = 2. $\underset{\{v_1, v_2\}}{\mathrm{M}}$ means that module M has observed the values v_1 and v_2 ($collected(M) = \{v_1, v_2\}$). Messages are depicted as arrows.

2.7 Machine "HybridGuarantees"

The distinction between faulty and non-faulty modules can be done in a more fine-grained manner. [10] introduces a hybrid fault model which distinguishes between arbitrarily faulty, manifest faulty and symmetrically faulty modules:

arbFaulty Modules which can send or drop message at their discretion. However, they cannot forge messages and will send a set which is a subset of the messages a non-faulty module would relay.

symFaulty The failure of the module is symmetric. If a message is changed or dropped, the same behaviour is presented to all potential recipients of the message. Since we can always rely on secure signatures, we can model this by having symFaulty nodes either drop a message or completely relay it. A symmetrically faulty module can choose its behaviour for any incoming message separately, however.

manifestFaulty Such a module always sends an erroneous message which can be—due to assumption A4—detected and, hence, discarded. A manifest faulty node is (with signatures) merely the corner case of a never-sending symmetrically faulty module.

In Fig. 10 we show the machine which implements this differentiation. In congruence to machine GUARANTEES there is also a technical counterpart introduced to conduct the proofs of agreement. We will omit details here, see Sect. 3. Some properties established in the refined machine no longer hold in this context and have to be adapted. For instance, it is now not any longer guaranteed that a message of length greater than card(*arbFaulty*) contains a non-faulty module; it merely contains one non-arbitrarily-faulty (i.e. symmetrically-faulty or non-faulty) one (invariant `ex_nonArbFaulty`). Such and similar adaptations have to be made to many invariants. The agreement proofs done earlier can therefore not simply be copied, but need to be redone. The experience gained in the simpler cases certainly helped in the challenge to discharge the new proof obligations, but proves could not be reused.

Please note that the event descriptions have now become significantly more complex. The different behaviour of *symFaulty*, *arbFaulty* and non-faulty nodes has to be reflected both in the initialisation and the round event. For the latter we use an indeterministic choice for the parameter *nonrec* which chooses a subset of the messages in *rec* with symmetrically faulty senders. Those messages will be dropped, not handled. For the remaining messages, symmetrically faulty modules behave like non-faulty ones.

2.8 Machine "ValueTables"

This refinement includes more information into the messages sent between modules. It allows the modules to build up a *value table* storing their knowledge of what value which module has originally received in round 1 (i.e. directly from the transmitter). This table allows majority or other votes on the multiset of originally sent values rather than only on the set of observed values.

It turns out, however, that we do not have the same termination constraints. While in the previous machine, we could guarantee agreement if $round \geq \mathrm{card}(faulty) + 1$, the protocol does *not* in general ensure that all non-faulty modules have built up identical tables in that round already. But we made the following observations:

1. For any voting[4] function, all non-faulty modules come to the same conclusion if $round \geq \mathrm{card}(faulty) + 1$ (invariant `agreement_voting`).

2. For $round \geq \mathrm{card}(faulty) + 2$ the protocol (finally) ensures that all non-faulty modules have identical tables.

3. In the case that the transmitter is faulty, the condition $round \geq \mathrm{card}(faulty) + 1$ suffices to have identical tables.

We use $round \geq \mathrm{card}(faulty \cup \{transmitter\}) + 1$ as the condition to have identical tables. Invariant `agreement` unifies observations 2 and 3.

2.8.1 Modified Problem Statement

The problem statement given in Sect. 1.1 is still valid. There are only two amendments:

1. The messages sent consist of a pair (module m, value v) indicating that m has originally (in round 1) received v. Modules can only pass on pairs or drop them, they can never defer or forge them.

2. In round 0 the transmitter distributes an initial message (m, v) to every module m. For a non-faulty transmitter v is equal to the intended value V_0, they may differ in case of a faulty transmitter.

[4]A voting is a function which selects a value from a non-empty multiset and a default value from the empty set.

MACHINE HybridGuarantees
REFINES Guarantees
SEES HybridContext
VARIABLES

```
rec
round
collected
```

INVARIANTS (`agreement` lifted)

`ex_nonArbFaulty` : $round \geq \mathrm{card}(arbFaulty) \Rightarrow$
$(\forall s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \Rightarrow (\exists x \cdot x \in l \land x \notin arbFaulty))$

`nonArb_collected` : $round \geq 1 \land transmitter \notin arbFaulty \Rightarrow$
$(\exists v \cdot collected = \mathtt{MODULE} \times \{\{v\}\}) \lor collected = \mathtt{MODULE} \times \{\varnothing\}$

`agreement` : $round \geq \mathrm{card}(arbFaulty) + 1 \Rightarrow$
$(\forall n, m \cdot n \notin faulty \land m \notin faulty \Rightarrow collected(n) = collected(m))$

EVENTS

Initialisation

inherits: `act3`

begin

`act1` : $rec, collected :|$
$\exists values \cdot values \in \mathtt{MODULE} \nrightarrow \mathtt{VALUE} \land$
$(transmitter \notin faulty \Rightarrow values = \mathtt{MODULE} \times \{V_0\}) \land$
$(transmitter \in symFaulty \Rightarrow values = \varnothing \lor (\exists v \cdot values = \mathtt{MODULE} \times \{v\})) \land$
$rec' = \{n \cdot n \in dom(values) \setminus \{transmitter\} \mid$
$(transmitter \mapsto n) \mapsto (\{transmitter\} \mapsto values(n))\} \land$
$collected' = (\mathtt{MODULE} \times \{\varnothing\}) \lhd\!\!- \{transmitter \mapsto ran(values)\}$

end

Event *ROUND* $\widehat{=}$
refines *ROUND* *(inherits:* `act2, act3`*)*

any

nonrec

where

`choice_nonrec` : $nonrec \subseteq \{l, v, s, r \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \land r \in symFaulty \mid$
$(s \mapsto r) \mapsto (l \mapsto v)\}$

then

`act1` : $rec :|$
let $RH = \{l, v, s, r, n \cdot (s \mapsto r) \mapsto (l \mapsto v) \in rec \setminus nonrec \land n \neq r \land n \notin l \mid$
$(r \mapsto n) \mapsto ((l \cup \{r\}) \mapsto v)\}$
in $rec' \subseteq RH \land$
$(\forall s_1, r_1, l_1, v_1 \cdot (s_1 \mapsto r_1) \mapsto (l_1 \mapsto v_1) \in RH \Rightarrow$
$(((s_1 \notin faulty \land v_1 \notin collected(s_1)) \lor s_1 \in symFaulty) \Rightarrow$
$(s_1 \mapsto r_1) \mapsto (l_1 \mapsto v_1) \in rec'))$

end

END

Figure 10: Machine HybridGuarantees

2.8.2 Modified Model

This refinement is a *representation change.* The set of messages rec is replaced by a set $msgs$ in which every element has one more component containing the original[5] receiver of the value. If this extra component is discarded (projection to the tuple without this component), this corresponds to the set rec of simpler messages:

$$\texttt{type_msgs:}\ msgs \in (\text{MODULE} \times \text{MODULE}) \leftrightarrow (\mathbb{P}(\text{MODULE}) \times \text{MODULE} \times \text{VALUE})$$

$$\texttt{glue_msgs_rec:}\ (\forall s, r, l, v \cdot ((s \mapsto r) \mapsto (l \mapsto v) \in rec \Leftrightarrow (\exists f \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs)))$$

Instead of a function $collected : \text{MODULE} \rightarrow \mathbb{P}(\text{VALUE})$ we have a value table function $valtable : \text{MODULE} \rightarrow (\text{MODULE} \nrightarrow \text{VALUE})$ which assigns to every module a partial table in which the knowledge about initially sent values is stored. For example $valtable(n)(m) = v$ means that module n knows that module m initially received value v. Since by assumption A4, no other values can be introduced, the range of that function is the set of seen values for a module:

$$\texttt{glue_valtable_collected:}\ \forall n \cdot collected(n) = ran(valtable(n))$$

The machine definitorial extension ValueTablesTech enriches the model by two variables *msgs_old* and *valtable_old* which hold the value of *msgs* and *valtable* of the previous round. The additional information made—again—the model and its proof obligations more complex and many of the results obtained earlier had to be redone. But again, the insight gained in the less complex situations helped to come up with the desired results here. Please see Fig. 11 for machine ValueTable.

2.9 Machine "ZA"

The last refinement in this chain is machine ZA which models the algorithm according to its presentation in [3]. The ZA algorithm is an algorithm for written messages which sends more messages than would be necessary to reach agreement. It is said it has a higher fall-back-security in case the signatures are not secure.

The algorithm is originally described as a recursive procedure with $ZA(n)$ referring to $ZA(n-1)$. For the purposes of this model within the event-based methodology of Event-B, we have made the rounds of the algorithm explicit and changed the modelling of the messages. While we did consider the history of a message to be a set so far, the ZA algorithm considers it an ordered sequence of modules, here an element of the set *ModuleList*. We did not need the properties of lists since we kept extra data like the sender, receiver, and first receiver separately. In ZA the representation of messages is changed to the variable *msgsZA* with

$$msgsZA \in (\text{MODULE} \times \text{MODULE}) \leftrightarrow (ModuleList \times \text{VALUE})\ .$$

Messages whose lists are permutations of one another come together to the same abstract representation, there is a "one-to-many" relationship between the elements in $msgs$ and $msgsZA$.

In ValueTable we explicitly specified the first receiver in the message tuple, this information can now be taken from the history sequence. In round 0 the receiver is the first receiver, in any subsequent round, it is the the first element $h(1)$ of the history h of a message. This is captured in the glueing invariant `glue_msgsZA_msgs`:

$$\begin{aligned}(round = 0 \Rightarrow (\forall s_0, r_0, h_0, v_0 \cdot (s_0 \mapsto r_0) \mapsto (h_0 \mapsto v_0) \in msgsZA \Rightarrow \\ (s_0 \mapsto r_0) \mapsto (ran(h_0) \mapsto r_0 \mapsto v_0) \in msgs)) \\ \wedge\ (round \geq 1 \Rightarrow (\forall s_1, r_1, h_1, v_1 \cdot (s_1 \mapsto r_1) \mapsto (h_1 \mapsto v_1) \in msgsZA \Rightarrow \\ (s_1 \mapsto r_1) \mapsto (ran(h_1) \mapsto h_1(1) \mapsto v_1) \in msgs))\end{aligned}$$

We have worked with sets up to now for good reasons: The Event-B methodology and the Rodin tool have considerable difficulties when it comes to handling sequences: they are not supported natively, but must be manually modelled in a context. We opted for modelling them as partial functions from natural numbers to modules. This is captured in the definition[6]

$$\texttt{type_moduleList:}\quad ModuleList = (\bigcup n \cdot n \in \mathbb{N} \mid 0\,..\,n \rightarrow \text{MODULE})$$

[5] we usually use f for *first* receiver

[6] The empty sequence is not in *ModuleList.* We have not included it as we do not need it for our purposes.

MACHINE ValueTables
REFINES HybridGuarantees
SEES HybridContext
VARIABLES

`msgs` `round` `valtable`

INVARIANTS (`agreement`, `agreement_voting` lifted)

`type_msgs` : $msgs \in (\text{MODULE} \times \text{MODULE}) \leftrightarrow (\mathbb{P}(\text{MODULE}) \times \text{MODULE} \times \text{VALUE})$

`glue_msgs_rec` : $(\forall s, r, l, v \cdot ((s \mapsto r) \mapsto (l \mapsto v) \in rec \Leftrightarrow (\exists f \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs)))$

`type_valtable` : $valtable \in \text{MODULE} \rightarrow (\text{MODULE} \nrightarrow \text{VALUE})$

`glue_valtable_collected` : $\forall n \cdot collected(n) = ran(valtable(n))$

`validity` : $\forall n \cdot valtable(n) \subseteq valtable(transmitter)$

`agreement` : $round \geq \text{card}(arbFaulty \cup \{transmitter\}) + 1 \Rightarrow$
$(\forall n, m \cdot n \notin faulty \wedge m \notin faulty \wedge m \neq transmitter \wedge n \neq transmitter \Rightarrow$
$valtable(n) = valtable(m))$

`agreement_voting` : $round \geq \text{card}(arbFaulty) + 1 \Rightarrow$
$(\forall n, m \cdot n \notin faulty \wedge m \notin faulty \wedge m \neq transmitter \wedge n \neq transmitter \Rightarrow$
$vote(valtable(n)) = vote(valtable(m)))$

EVENTS

Initialisation *(inherits:* `act3`*)*

begin

with

`rec'` : $rec' = \{s, r, l, f, v \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs' \mid (s \mapsto r) \mapsto (l \mapsto v)\}$

collected' : $collected' = (\lambda n \cdot n \in \text{MODULE} \mid ran(valtable'(n)))$

`act1` : $msgs, valtable$

$\exists values \cdot values \in \text{MODULE} \nrightarrow \text{VALUE} \wedge$
$(transmitter \notin faulty \Rightarrow values = \text{MODULE} \times \{V_0\}) \wedge$
$(transmitter \in symFaulty \Rightarrow values = \varnothing \vee (\exists v \cdot values = \text{MODULE} \times \{v\})) \wedge$
$msgs' = \{n \cdot n \in dom(values) \setminus \{transmitter\} \mid$
$(transmitter \mapsto n) \mapsto (\{transmitter\} \mapsto n \mapsto values(n))\} \wedge$
$valtable' = (\text{MODULE} \times \{\varnothing\}) \Leftarrow \{transmitter \mapsto \{transmitter\} \lhd values\}$

end

Event $ROUND \mathrel{\widehat{=}}$

refines $ROUND$ *(inherits:* `act3`*)*

any

$nonrecmsgs$

where

`choice_nonrecmsgs` : $nonrecmsgs \subseteq \{s, r, l, f, v \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs \wedge r \in symFaulty \mid (s \mapsto r) \mapsto (l \mapsto f \mapsto v)\}$

with

`rec'` : $rec' = \{s, r, l, f, v \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs' \mid (s \mapsto r) \mapsto (l \mapsto v)\}$

nonrec : $nonrec = \{s, r, l, v, f \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in nonrecmsgs \wedge (\forall f_1 \cdot (s \mapsto r) \mapsto (l \mapsto f_1 \mapsto v) \in msgs \Rightarrow (s \mapsto r) \mapsto (l \mapsto f_1 \mapsto v) \in nonrecmsgs) \mid (s \mapsto r) \mapsto (l \mapsto v)\}$

then

`act1` : $msgs :\mid \text{let } M = \{l, f, v, s, r, n \cdot$
$(s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs \setminus nonrecmsgs \wedge n \neq r \wedge n \notin l \mid$
$(r \mapsto n) \mapsto ((l \cup \{r\}) \mapsto f \mapsto v)\} \text{ in}$
$msgs' \subseteq M \wedge$
$(\forall l_1, f_1, v_1, s_1, r_1 \cdot (s_1 \mapsto r_1) \mapsto (l_1 \mapsto f_1 \mapsto v_1) \in M \Rightarrow$
$(((s_1 \notin faulty \wedge f_1 \mapsto v_1 \notin valtable(s_1)) \vee s_1 \in symFaulty) \Rightarrow$
$(s_1 \mapsto r_1) \mapsto (l_1 \mapsto f_1 \mapsto v_1) \in msgs'))$

`act2` : $valtable := \lambda n \cdot n \in \text{MODULE} \mid valtable(n) \Leftarrow$
$\{s, l, f, v \cdot (s \mapsto n) \mapsto (l \mapsto f \mapsto v) \in msgs \mid f \mapsto v\}$

end

END

Figure 11: Machine ValueTables

in context ModuleList (Fig. 12) which was introduced for that reason. This context contains also helpful theorems which were needed when proving the correctness of ZA. Please note that the definitions are not polymorphous but contain hard coded the carrier set `MODULE`. For another type, the definitions (and proofs) would have to be repeated.

When dealing with this formalisation of sequences, the user has to manually prove that concatenating a value to a list results in a list again at very many places. Despite the fact that nothing "new" was introduced in this machine, and agreement and other important properties could directly be inherited from ValueTables, the proofs were considerably complex. The lacking support for sequences in Rodin can hardly be compensated for manually.

CONTEXT ModuleList
EXTENDS Context
CONSTANTS

`ModuleList`
`fstrec`

AXIOMS

`type_moduleList` : $ModuleList = (\bigcup n \cdot n \in \mathbb{N} \mid 0 \,..\, n \rightarrow MODULE)$

`thm1` : $ModuleList \subseteq \mathbb{N} \mathrel{+\!\!\!\rightarrow} MODULE$

`thm2` : $\forall ml \cdot ml \in ModuleList \Rightarrow (\exists n \cdot n \in \mathbb{N} \wedge dom(ml) = 0 \,..\, n)$

`type_fstrec` : $fstrec \in (MODULE \times ModuleList) \rightarrow MODULE$

`def_fstrec` : $\forall r, h \cdot h \in ModuleList \Rightarrow ((1 \in dom(h) \Rightarrow fstrec(r \mapsto h) = h(1)) \wedge (1 \notin dom(h) \Rightarrow fstrec(r \mapsto h) = r))$

`fstrecEmpty` : $\forall r, h, m \cdot h \in ModuleList \wedge 1 \notin dom(h) \Rightarrow fstrec(r \mapsto h \mathbin{\lhd\!\!\!-} \{1 \mapsto m\}) = m$

`fstrecNonEmpty` : $\forall r, h, m, k \cdot h \in ModuleList \wedge k \geq 1 \wedge dom(h) = 0 \,..\, k \Rightarrow fstrec(r \mapsto h \mathbin{\lhd\!\!\!-} \{k+1 \mapsto m\}) = h(1)$

`concatenation` : $\forall ml, m, k \cdot ml \in ModuleList \wedge dom(ml) = 0 \,..\, k \Rightarrow ml \mathbin{\lhd\!\!\!-} \{k+1 \mapsto m\} \in ModuleList$

`samefstrecValue` :
$\forall m, h, n, k \cdot n \in MODULE \wedge m \in MODULE \wedge h \in ModuleList \wedge dom(h) = 0 \,..\, k \Rightarrow fstrec(m \mapsto h) = fstrec(n \mapsto h \mathbin{\lhd\!\!\!-} \{k+1 \mapsto m\})$

END

Figure 12: Context ModuleList

2.10 Machine "Roundless"

We end the exploration of the ZA protocol now and turn our attention towards another algorithm: Signed Messages (SM) as it was proposed in [5]. In contrast to most other protocols, this protocol has not been presented as a strictly round-based, synchronous, recursive algorithm but as a form of reactive system:

> If [a] lieutenant receives a message then [...] he sends [other] messages.

To model this as an refinement (starting at Guarantees) is an interesting task since it changes the approach to the algorithm fundamentally. While up to now we concentrated on seeing the system with all modules in each event we should now come up with events whose focus is the behaviour of one single module and one single incoming message.

The main two changes introduced in machine Roundless (Fig. 14) are: Firstly, the machine's variables are no longer bound to the current round but may hold messages of various rounds, and secondly, we add a new event Process acting as a generalisation of the processing of one particular message by its receiver.

MACHINE ZA
REFINES ValueTables
SEES HybridContext, ModuleList
VARIABLES

$msgsZA$ `round` `valtable`

INVARIANTS

`type_msgsZA` : $msgsZA \in (\mathrm{MODULE} \times \mathrm{MODULE}) \leftrightarrow (ModuleList \times \mathrm{VALUE})$

`domainOfHistory` : $\forall s, r, h, v \cdot (s \mapsto r) \mapsto (h \mapsto v) \in msgsZA \Rightarrow dom(h) = 0 \,..\, round$

`glue_msgsZA_msgs` : $(round = 0 \Rightarrow (\forall s_0, r_0, h_0, v_0 \cdot (s_0 \mapsto r_0) \mapsto (h_0 \mapsto v_0) \in msgsZA \Rightarrow (s_0 \mapsto r_0) \mapsto (ran(h_0) \mapsto r_0 \mapsto v_0) \in msgs)) \wedge (round \geq 1 \Rightarrow (\forall s_1, r_1, h_1, v_1 \cdot (s_1 \mapsto r_1) \mapsto (h_1 \mapsto v_1) \in msgsZA \Rightarrow (s_1 \mapsto r_1) \mapsto (ran(h_1) \mapsto h_1(1) \mapsto v_1) \in msgs))$

`glue_msgZA_fstrec` : $msgs = \{s, r, h, v \cdot (s \mapsto r) \mapsto (h \mapsto v) \in msgsZA \mid (s \mapsto r) \mapsto (ran(h) \mapsto fstrec(r \mapsto h) \mapsto v)\}$

`last_is_sender` : $\forall s, r, h, v \cdot (s \mapsto r) \mapsto (h \mapsto v) \in msgsZA \Rightarrow h(round) = s$

EVENTS

Initialisation

begin

with

`msgs'` : $msgs' = \{s, r, h, v \cdot (s \mapsto r) \mapsto (h \mapsto v) \in msgsZA' \mid (s \mapsto r) \mapsto (ran(h) \mapsto r \mapsto v)\}$

`act1` : $msgsZA, valtable :\mid \exists values \cdot values \in \mathrm{MODULE} \nrightarrow \mathrm{VALUE} \wedge (transmitter \notin faulty \Rightarrow values = \mathrm{MODULE} \times \{V_0\}) \wedge (transmitter \in symFaulty \Rightarrow values = \varnothing \vee (\exists v \cdot values = \mathrm{MODULE} \times \{v\})) \wedge msgsZA' = \{n \cdot n \in dom(values) \setminus \{transmitter\} \mid (transmitter \mapsto n) \mapsto (\{0 \mapsto transmitter\} \mapsto values(n))\} \wedge valtable' = (\mathrm{MODULE} \times \{\varnothing\}) \Leftarrow \{transmitter \mapsto (\{transmitter\} \lhd values)\}$

`act3` : $round := 0$

end

Event $ROUND \mathrel{\widehat{=}}$
refines $ROUND$

any

$nonrecmsgsZA$

where

`choice_nonrecmsgsZA` : $nonrecmsgsZA \subseteq \{s, r, h, v \cdot (s \mapsto r) \mapsto (h \mapsto v) \in msgsZA \wedge r \in symFaulty \mid (s \mapsto r) \mapsto (h \mapsto v)\}$

with

`nonrecmsgs` : $nonrecmsgs = msgs \setminus \{s, r, h, v \cdot (s \mapsto r) \mapsto (h \mapsto v) \in msgsZA \setminus nonrecmsgsZA \mid (s \mapsto r) \mapsto (ran(h) \mapsto fstrec(r \mapsto h) \mapsto v)\}$

`msgs'` : $msgs' = \{s, r, h, v \cdot (s \mapsto r) \mapsto (h \mapsto v) \in msgsZA' \mid (s \mapsto r) \mapsto (ran(h) \mapsto fstrec(r \mapsto h) \mapsto v)\}$

then

`act1` : $msgsZA :\mid \text{let } M = \{h, v, s, r, n \cdot (s \mapsto r) \mapsto (h \mapsto v) \in msgsZA \setminus nonrecmsgsZA \wedge n \neq r \wedge n \notin ran(h) \mid (r \mapsto n) \mapsto ((h \Leftarrow \{round + 1 \mapsto r\}) \mapsto v)\} \text{ in } msgsZA' \subseteq M \wedge (\forall h_1, v_1, s_1, r_1 \cdot (s_1 \mapsto r_1) \mapsto (h_1 \mapsto v_1) \in M \Rightarrow (((s_1 \notin faulty \wedge fstrec(r_1 \mapsto h_1) \mapsto v_1 \notin valtable(s_1)) \vee s_1 \in symFaulty) \Rightarrow (s_1 \mapsto r_1) \mapsto (h_1 \mapsto v_1) \in msgsZA'))$

`act2` : $valtable := (\lambda n \cdot n \in MODULE \mid valtable(n) \Leftarrow \{s, h, v \cdot (s \mapsto n) \mapsto (h \mapsto v) \in msgsZA \mid fstrec(n \mapsto h) \mapsto v\})$

`act3` : $round := round + 1$

end

END

Figure 13: Machine ZA

Two new variables *msgPast* and *msgPool* with

$$\begin{array}{rcl} msgPast & \in & (\texttt{Module} \times \texttt{Module}) \leftrightarrow (\mathbb{P}(\texttt{Module}) \times \texttt{Value}) \\ msgPool & \in & (\texttt{Module} \times \texttt{Module}) \leftrightarrow (\mathbb{P}(\texttt{Module}) \times \texttt{Value}) \end{array}$$

are introduced to hold all messages of various rounds. When a message is processed by its receiver it is removed from *msgPool* and added to *msgPast*. Newly created messages are then added to *msgPool*. The two glueing invariants

$$\texttt{glue_msgPool_rec:} \quad rec = \{s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in msgPool \cup msgPast \wedge \text{card}(l) = round + 1 \mid (s \mapsto r) \mapsto (l \mapsto v)\}$$

$$\texttt{glue_collected_msgPast:} \quad \forall n \cdot collected(n) = \{s, l, v \cdot (s \mapsto n) \mapsto (l \mapsto v) \in msgPast \wedge \text{card}(l) \leq round \mid v\}$$

show that *rec* is the subset of $msgPool \cup msgPast$ of messages of the current round ($\text{card}(l) = round + 1$). The variable *collected* is reproduced by examining the values of all messages in *msgPast*.

It is interesting to see that event Round has no more substitutional part of its own (apart from the $round := round + 1$ inherited from its abstract ancestor). The actual message processing now in one or more executions of event Process, and Round is a mere change of perspective which has no influence on the message data. We have to provide, however, witnesses to refine the event of machine HybridGuarantees which instantiate the post state rec' and the choice of dropped messages *nonrec*. A guard `all_handled` ensures that the round event is only taken when all messages of the current round have been processed (and possibly some more). Without it the refinement would be incorrect.

Event Process is newly introduced and, therefore, must not have effects on the variables of the refined machine, i.e. it must refine the "skip" event. Despite the fact that *msgPool* and *msgPast* are modified by this event, the values of *collected* and *rec* which are calculated from these sets, do not change. The processing is kept general, all possible treatments are put into one event, a later refinement can differentiate between different cases here.

This machine is obviously no longer in accordance with A1. As we have announced earlier, we now drop this assumption in favour of A6.

2.11 Machine "SM"

The last machine in our refinement hierarchy is depicted in Fig. 15. It does neither modify the Initialisation nor the Round event but refines the Process event in two different ways: One event (Process nonfaulty) models the processing of a sane module. All messages that could be relayed are relayed. The other (Process drop) models the case in which a module does not at all react to incoming messages. These are two corner cases of the possible behaviour of modules and one could add more such refinements if needed.

This machine is different because it does not (like most other machines) introduce new behaviour or refine data structures. It merely differentiates an abstract event (Process in Roundless) into more concrete instances. It is also the only machine for which all proofs were simple enough to be proved automatically by the built-in solvers of Rodin.

The machine was designed to model the algorithm SM as presented in [5]

Algorithm SM(m)[7]:
Initially $collected(i) = \varnothing$.

1. The transmitter signs and sends its value to every module (but itself).
2. For each i:
   (a) If module i receives a message of the form $v : 0$ from the transmitter and it has not yet received any order, then
      i. it lets $collected(i) = \{v\}$
      ii. it sends the message $v : 0 : i$ to every other module.
   (b) If module i receives a message of the form $v : 0 : j_1 : \ldots : j_k$ and v is not in the set $collected(i)$, then

[7] adapted to our definitions

MACHINE ROUNDLESS
REFINES GUARANTEES
SEES CONTEXT
VARIABLES

msgPool msgPast round

INVARIANTS

type_msgPool : $msgPool \in (\text{MODULE} \times \text{MODULE}) \leftrightarrow (\mathbb{P}(\text{MODULE}) \times \text{VALUE})$

type_msgPast : $msgPast \in (\text{MODULE} \times \text{MODULE}) \leftrightarrow (\mathbb{P}(\text{MODULE}) \times \text{VALUE})$

messages_content : $\forall s, r, l, v \cdot s \mapsto r \mapsto (l \mapsto v) \in msgPast \cup msgPool \wedge l \neq \varnothing \Rightarrow \text{finite}(l) \wedge s \in l \wedge r \notin l$

msg_implies_msg : $\forall s, r, l, v \cdot s \mapsto r \mapsto (l \mapsto v) \in msgPast \cup msgPool \wedge card(l) > 1 \Rightarrow (\exists n \cdot (n \mapsto s) \mapsto (l \setminus \{s\} \mapsto v) \in msgPast \wedge n \in l \wedge n \neq s)$

pool_round : $\forall s, r, l, v \cdot s \mapsto r \mapsto (l \mapsto v) \in msgPool \Rightarrow card(l) > round$

glue_msgPool_rec : $rec = \{s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in msgPool \cup msgPast \wedge card(l) = round + 1 \mid (s \mapsto r) \mapsto (l \mapsto v)\}$

glue_collected_msgPast : $\forall n \cdot collected(n) = \{s, l, v \cdot (s \mapsto n) \mapsto (l \mapsto v) \in msgPast \wedge card(l) \leq round \mid v\}$

guarantee : $\forall s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in msgPast \wedge r \notin faulty \wedge \neg(\exists s_1, l_1 \cdot (s_1 \mapsto r) \mapsto (l_1 \mapsto v) \in msgPast \wedge card(l_1) \leq round) \Rightarrow \{n \cdot n \neq r \wedge n \notin l \mid (r \mapsto n) \mapsto (l \cup \{r\} \mapsto v)\} \subseteq msgPool \cup msgPast$

EVENTS

Initialisation

begin

with

collected′ : $collected' = (\text{MODULE} \times \{\varnothing\}) \lhd\!\!- \{transmitter \mapsto ran(ran(msgPool'))\}$

rec′ : $rec' = msgPool'$

act1 : $msgPool, msgPast :\mid \exists values \cdot values \in \text{MODULE} \twoheadrightarrow \text{VALUE} \wedge (transmitter \notin faulty \Rightarrow values = \text{MODULE} \times \{V_0\}) \wedge msgPool' = \{n \cdot n \in dom(values) \setminus \{transmitter\} \mid (transmitter \mapsto n) \mapsto (\{transmitter\} \mapsto values(n))\} \wedge msgPast' = \{transmitter \mapsto transmitter\} \times (\{\varnothing\} \times ran(ran(msgPool')))$

act2 : $round := 0$

end

Event *ROUND* $\widehat{=}$ **refines** *ROUND*

when

all_handled : $\forall s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in msgPool \Rightarrow card(l) > round + 1$

with

rec′ : $rec' = \{s, r, l, v \cdot (s \mapsto r) \mapsto (l \mapsto v) \in msgPool \cup msgPast \wedge card(l) = round + 2 \mid (s \mapsto r) \mapsto (l \mapsto v)\}$

then

act1 : $round := round + 1$

end

Event *PROCESS* $\widehat{=}$

any *output* *s* *r* *l* *v*

where

grd1 : $(s \mapsto r) \mapsto (l \mapsto v) \in msgPool$

out1 : $output \subseteq \{n \cdot n \neq r \wedge n \notin l \mid (r \mapsto n) \mapsto (l \cup \{r\} \mapsto v)\}$

out2 : $r \notin faulty \wedge \neg(\exists s_1, l_1 \cdot (s_1 \mapsto r) \mapsto (l_1 \mapsto v) \in msgPast \wedge card(l_1) \leq round) \Rightarrow output = \{n \cdot n \neq r \wedge n \notin l \mid (r \mapsto n) \mapsto (l \cup \{r\} \mapsto v)\}$

then

act1 : $msgPool := (msgPool \setminus \{(s \mapsto r) \mapsto (l \mapsto v)\}) \cup output$

act2 : $msgPast := msgPast \cup \{(s \mapsto r) \mapsto (l \mapsto v)\}$

end

END

Figure 14: Machine ROUNDLESS

i. it adds v to $collected(i)$;

ii. if $k < m$, then it sends the message $v : 0 : j_1 : \ldots : j_k : i$ to every module other than $j_1, \ldots, j_k$.

(c) For each i: When module i will receive no more messages, it obeys the order obtained from $collected(i)$.

This roundless algorithm is, unfortunately, *not* a refinement of the stack of machines as they were developed so far. If (in SM) a node handles a message of length $\text{card}(l) = N$, it records its value as seen and does not relay any further messages with this value, i.e., a handled message could prevent the sending of a message in a round $round < N$ which is forbidden in our formalisation as the Roundless algorithms have to emulate the effects of the roundbased descriptions. We mend this problem by slightly changing the algorithm:

2. (b) If module i receives a message of the form $v : 0 : j_1 : \ldots : j_k$ and i has not yet received a shorter message with the value v, then ...

A non-faulty node now relays every message unless it has experienced a message with the same value *and a shorter history.*

3 Important Properties

The properties for which we wanted to provide proofs are formulated as invariants in the various machines. We also came up with a number of invariants which are lemmata which helped finding proofs for the more complicated properties. The two main ideas we haven proved are *validity* and *agreement*.

After summarising the different formulas which belong to the two categories, we list a number of intermediate results and show how they can be combined to gain a proof for an agreement property.

Validity The notion of what validity is varies in the literature. We understand it as: "If the transmitter is non-faulty, then all modules ascribe the value V_0 to the transmitter." The following properties belong to that category:

- MessagesSigned.`no_new_vals` ensures that any value that ever appears in a message has been originally observed by the transmitter already. It is a direct consequence of A4.

$$\forall s, r, v \cdot (s \mapsto r) \mapsto v \in messages \Rightarrow v \in collected(transmitter)$$

- MessagesSigned.`no_new_vals2` describes the same fact for collected values. Any value ever collected by a module must also have been observed by the transmitter.

$$\forall n \cdot collected(n) \subseteq collected(transmitter)$$

- Guarantees.`validity` is the intended validity property.

$$round \geq 1 \land transmitter \notin faulty \Rightarrow (\forall n \cdot collected(n) = \{V_0\})$$

- HybridGuarantees.`nonArb_collected` contains a more general property. If the transmitter is not arbitrarily faulty, all modules see either the same single value v or no module receives any value at all.

$$round \geq 1 \land transmitter \notin arbFaulty \Rightarrow$$
$$(\exists v \cdot collected = \texttt{MODULE} \times \{\{v\}\}) \lor collected = \texttt{MODULE} \times \{\varnothing\}$$

Agreement Agreement means: "Any two non-faulty modules agree on the value ascribed to the transmitter." Agreement is reached only after a certain number of rounds. Sect. 1.3 discusses why the parameter of protocols in the literature (like ZA(r)) only seem to be smaller by one in comparison to the bounds of this formalisation. There are agreement theorems for three stages[8] of abstraction:

[8] always defined in the corresponding technical definitorial extension

MACHINE SM
REFINES ROUNDLESS
SEES HYBRIDCONTEXT
VARIABLES

msgPool

msgPast

round

EVENTS

Initialisation

(inherits: act1, act2*)*

begin
end

Event $PROCESS_NONFAULTY \widehat{=}$
refines $PROCESS$

any

$s \quad r \quad l \quad v$

where

grd1 : $(s \mapsto r) \mapsto (l \mapsto v) \in msgPool$

with

output : $output = \{n \cdot n \neq r \wedge n \notin l \mid (r \mapsto n) \mapsto (l \cup \{r\} \mapsto v)\}$

then

act1 : $msgPool := (msgPool \setminus \{(s \mapsto r) \mapsto (l \mapsto v)\})$
$\cup \{n \cdot n \neq r \wedge n \notin l \mid (r \mapsto n) \mapsto (l \cup \{r\} \mapsto v)\}$

act2 : $msgPast := msgPast \cup \{(s \mapsto r) \mapsto (l \mapsto v)\}$

end

Event $PROCESS_DROP \widehat{=}$
refines $PROCESS$

any

$s \quad r \quad l \quad v$

where

grd1 : $(s \mapsto r) \mapsto (l \mapsto v) \in msgPool$

grd2 : $r \in faulty$

with

output : $output = \varnothing$

then

act1 : $msgPool := msgPool \setminus \{(s \mapsto r) \mapsto (l \mapsto v)\}$

act2 : $msgPast := msgPast \cup \{(s \mapsto r) \mapsto (l \mapsto v)\}$

end

END

Figure 15: Machine SM

- GUARANTEES.`agreement`:

$$round \geq \text{card}(faulty) + 1 \Rightarrow (\forall n, m \cdot n \notin faulty \land m \notin faulty \Rightarrow collected(n) = collected(m))$$

- HYBRIDGUARANTEES.`agreement` relaxes the premise. The number of rounds must only exceed the number of arbitrarily faulty modules.

$$round \geq \text{card}(arbFaulty) + 1 \Rightarrow (\forall n, m \cdot n \notin faulty \land m \notin faulty \Rightarrow collected(n) = collected(m))$$

- If we store value tables instead of only values, the condition is more strict in VALUETABLES.`agreement`. The number of rounds needs to exceed $\text{card}(arbFaulty \cup \{transmitter\})$.

$$round \geq \text{card}(arbFaulty \cup \{transmitter\}) + 1 \Rightarrow \\ (\forall n, m \cdot n \notin faulty \land m \notin faulty \Rightarrow valtable(n) = valtable(m))$$

- If we are only interested in the result of a voting function, invariant VALUETABLES.`agreement_voting` allows us to relax the bound to the previously established $\text{card}(arbFaulty) + 1$:

$$round \geq \text{card}(arbFaulty) + 1 \Rightarrow \\ (\forall n, m \cdot n \notin faulty \land m \notin faulty \land m \neq transmitter \land n \neq transmitter \Rightarrow \\ vote(valtable(n)) = vote(valtable(m)))$$

  See also Sect. 3.3 for a discussion of this invariant.

3.1 Selected lemmata of ValueTables

To give the reader an impression of the nature of intermediate propositions that were needed for the deduction, we list here a few lemmata which helped in the process of proving validity and agreement. They represent only the important intermediate results and make up about half of the invariants provided in the technical machine. We will confine ourselves to the most important lemmata from the most detailed machine VALUETABLES, other machines have similar properties. The next section will then present an exemplary proof using these lemmata:

- `nonfaulty_broadcast` states that if a first-receiver-value-pair has been observed by a non-faulty node n different from the transmitter in a previous round ($f \mapsto v \in valtable_{old}(n)$), then n relays it correctly and the pair is known to every module afterwards.

$$\forall f, v \cdot (\exists n \cdot n \notin faulty \land n \neq transmitter \land (f \mapsto v) \in valtable_{old}(n)) \Rightarrow \\ (\forall m \cdot (f \mapsto v) \in valtable(m))$$

- `symfaulty_broadcast` describes a similar effect for symmetrically faulty modules: If in the past a symmetrically faulty module n decided to pass on a message (i.e. n is in the history: $n \in l$), then n passed it on to every module which had not yet seen the pair $f \mapsto v$.

$$\forall s, r, l, f, v, n \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs_{old} \land \\ n \in l \land n \in symFaulty \land n \neq transmitter \Rightarrow \\ (\forall m \cdot (f \mapsto v) \in valtable(m))$$

- `new_valtable_implies_msgs` observes: If a first-receiver-value-pair $f \mapsto v$ has recently been added to the valtable of a non-faulty node n (i.e. $f \mapsto v \in valtable(n) \setminus valtable_{old}(n)$), then for every module m we known that it either knows already about the value ($f \mapsto v \in valtable(m)$) or a message has been sent to it from n ($\exists l \cdot (n \mapsto m) \mapsto (l \mapsto f \mapsto v) \in msgs$).

$$\forall n \cdot n \notin faulty \land n \neq transmitter \Rightarrow (\forall f, v \cdot f \mapsto v \in valtable(n) \setminus valtable_{old}(n) \Rightarrow \\ (\forall m \cdot f \mapsto v \in valtable(m) \lor (\exists l \cdot (n \mapsto m) \mapsto (l \mapsto f \mapsto v) \in msgs)))$$

- `msgs_msgs_old` states that after the first round, every message in $msgs$ must have a cause, a message of a certain form sent in the previous round, which justifies its existence. This is not the case for the messages initially sent by the transmitter in round 0.

$$round \geq 1 \Rightarrow (\forall s, r, l, f, v \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs \Rightarrow (\exists n \cdot (n \mapsto s) \mapsto (l \setminus \{s\} \mapsto f \mapsto v) \in msgs_{old}))$$

- `ex_nonArbFaulty2` uses a simple cardinality argument: The length of the history (by construction) of the last round is *round*. Hence, we have $\text{card}(l) > \text{card}(\mathit{arbFaulty})$, which obviously implies the conclusion.

$$round \geq \text{card}(\mathit{arbFaulty}) + 1 \Rightarrow (\forall s, r, l, f, v \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs_{old} \Rightarrow (\exists x \cdot x \in l \land x \notin \mathit{arbFaulty}))$$

- `ex_nonArbFaulty_nontrans2` is very similar to the last property. If we increase the required *round* by one, and, hence, the length of histories, we may also assume a non-arb-faulty module in any message history which is not the transmitter.

$$round \geq \text{card}(\mathit{arbFaulty}) + 2 \Rightarrow (\forall s, r, l, f, v \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs_{old} \Rightarrow (\exists x \cdot x \in l \land x \notin \mathit{arbFaulty} \land x \neq \mathit{transmitter}))$$

- `msg_old_implies_valtable_old` captures the simple observation that all modules that have received a message before the last round ($n \in l$) had registered the according entry to their value table already in the last round.

$$\forall n, s, r, l, f, v \cdot (s \mapsto r) \mapsto (l \mapsto f \mapsto v) \in msgs_{old} \land n \in l \Rightarrow (f \mapsto v) \in valtable_{old}(n)$$

3.2 Sketch for the proof of `agreement_subset`

This section gives a transcript of the major steps taken to prove the invariant `agreement_subset`

$$round \geq card(\mathit{arbFaulty} \cup \{\mathit{transmitter}\}) + 1 \Rightarrow (\forall n, m \cdot n \notin \mathit{faulty} \land m \notin \mathit{faulty} \land n \neq \mathit{transmitter} \land m \neq \mathit{transmitter} \Rightarrow valtable(n) \subseteq valtable(m))$$

in ValueTableTech. The proof for similar properties in other machines runs analogously.

We present it to show that the structure of a complex Event-B proof can be subdivided (using lemmata) in a very similar fashion to in which one would do a pen-and-paper proof. The proof references to invariants established in this machine or in more abstract machines. They are listed in the previous section. Invariant `agreement` (with $=$ instead of $\subseteq$) is a direct consequence of this property due to the symmetric nature of the invariant.

Given that

- $round \geq \text{card}(\mathit{arbFaulty} \cup \{\mathit{transmitter}\}) + 1$,
- $n, m \notin \mathit{faulty}$,
- $n, m \neq \mathit{transmitter}$, and
- $x \mapsto x_0 \in valtable(n)$,

we show that $x \mapsto x_0 \in valtable(m)$.

Case Distinction

- $x \mapsto x_0 \in valtable_{old}(n)$[9] ... value has already been observed earlier

  `nonfaulty_broadcast` $\rightsquigarrow \forall m \cdot x \mapsto x_0 \in valtable(m)$
  If the message has already been received by the non-faulty n in the last round, the module has broadcast it to any other node. The message has been received in this round by any node which has not yet seen it, also by m (though it might have already known of it of course). ◆

[9] Please keep in mind that the technical extensions of the machines keep a copy of their variables (suffixed *old*) which holds the values of the previous round. (cf. Sect. 2.6.1)

- $x \mapsto x_0 \notin valtable_{old}(n)$ ... value has not been observed yet.

  `new_valtable_implies_msgs` $\leadsto \forall m \cdot (x \mapsto x_0 \in valtable(m) \vee \exists l \cdot (n \mapsto m) \mapsto (l \mapsto x \mapsto x_0) \in msgs)$
  The message was not seen by n before. That implies that n has received an according message recently. Being non-faulty, module n has to broadcast the message to all modules which are ignorant of this module-value-pair. Hence, we can assume that there is a message $(n \mapsto m) \mapsto (l \mapsto x \mapsto x_0) \in msgs$ (0).

  `msgs_msgs_old` $\leadsto \exists n_0 \cdot (n_0 \mapsto n) \mapsto (l \setminus \{n\} \mapsto x \mapsto x_0) \in msgs_{old}$ (1)
  Message (0) from n to m in this round can only arise, if there has been a message (1) in the previous round which started in some node n_0 and was addressed to n.

  Case distinction

  - $transmitter \in arbFaulty$ ... the transmitter is arb. faulty (2)
    $arbFaulty = arbFaulty \cup \{transmitter\} \leadsto round \geq \mathrm{card}(arbFaulty) + 1$
    `ex_non_arbFaulty2` $\leadsto x_1 \in l \setminus \{n\} \wedge x_1 \notin arbFaulty$ (3)
    Since for a message in $msgs_{old}$ we have that the length of the history equals the round number (`set_card2`), message (1) has sufficient length to include a non-arb-faulty module x_1 in its history $l \setminus \{n\}$. Notice, that x_1 cannot be the transmitter, because the latter is arbitrarily faulty while x_1 is not.
    `msg_old_implies_valtable_old` $\leadsto x \mapsto x_0 \in valtable_{old}(x_1)$ (4)
    Any module which appears in the history of a message has recorded the message's value in its value table. Message (1) implies that $x \mapsto x_0$ has been recorded in the table of x_1.
    Case distinction (5)
    * $x_1 \in symFaulty$ ... x_1 is symmetrically faulty,
      `symFaulty_broadcast` $\leadsto \forall m \cdot x \mapsto x_0 \in valtable(m)$
      x_1 is in the history of message (1) due to (3). That implies that it has acted as a sender and, due to the symmetry assumption, has relayed the message to all modules that have not yet seen the value, hereby ensuring everyone knows about it. ◆
    * $x_1 \notin symFaulty$ ... x_1 is not symmetrically faulty, i.e. $x_1 \notin faulty$ (because of (3))
      `nonfaulty_broadcast` $\leadsto \forall m \cdot x \mapsto x_0 \in valtable(m)$
      Since x_1 is non-faulty and has received the entry $x \mapsto x_0$ either in the last round or before that (4), it has broadcast the value to ensure everyone knows about it. ◆
  - $transmitter \notin arbFaulty$ ... the transmitter is not arbitrarily faulty
    $\mathrm{card}(arbFaulty \cup \{transmitter\}) = \mathrm{card}(arbFaulty) + \mathrm{card}(\{transmitter\}) \leadsto$
    $round \geq \mathrm{card}(arbFaulty) + 2$
    `ex_nonArbFaulty_nontrans2` $\leadsto x_1 \in l \setminus \{n\} \wedge x_1 \notin arbFaulty \wedge x_1 \neq transmitter$ (6)
    In this case the message (1) even has sufficient length to include a non-faulty module x_1 different to the transmitter in its history $l \setminus \{n\}$.
    `msg_old_implies_valtable_old` $\leadsto x \mapsto x_0 \in valtable_{old}(x_1)$ (7)
    Any module which appears in the history of a message has recorded the message's value in its value table. Message (1) implies that $x \mapsto x_0$ has been recorded in the table of x_1.
    The situation is now the same as in (5): $x_1 \neq transmitter$ because of (6) and $x \mapsto x_0 \in valtable_{old}(x_1)$ from (7). ◆

■

3.3 On the invariant `agreement_voting`

The result which was achieved for ValueTables in `agreement` was not satisfactory. It states that, in the case of a non-faulty transmitter $round \geq \mathrm{card}(arbFaulty \cup \{transmitter\}) + 1$ has to be satisfied to guarantee agreement on the value tables amongst the non-faulty modules. The agreement definition usually used (like in [3]) states that non-faulty modules agree on the value ascribed to the transmitter for $round \geq \mathrm{card}(arbFaulty) + 1$.

Since this "ascription" is done by a voting function $vote : (\texttt{MODULE} \nrightarrow \texttt{VALUE}) \rightarrow \texttt{VALUE}$ we hope that we can lower the bound for an agreement invariant on the result of voting function as in `agreement_voting`:

$$\begin{aligned}
&round \geq \mathrm{card}(arbFaulty) + 1 \Rightarrow \\
&\quad (\forall n, m \cdot n \notin faulty \land m \notin faulty \land m \neq transmitter \land n \neq transmitter \Rightarrow \\
&\qquad vote(valtable(n)) = vote(valtable(m)))
\end{aligned}$$

The context VotingContext where *vote* is defined, postulates only one requirement to that function:

$$\forall f \cdot f \in \texttt{MODULE} \nrightarrow \texttt{VALUE} \land ran(f) \neq \varnothing \Rightarrow vote(f) \in ran(f) \ ,$$

saying that if a table is not empty, the chosen value must be taken from the table.

Let us now first assume that $transmitter \in arbFaulty$. Then $faulty = faulty \cup \{transmitter\}$ holds and the premisses of `agreement` and `agreement_voting` are equal. Thus, the value tables are equal, thus, the voting results. In the case that $transmitter \notin arbFaulty$, we can use the proposition HybridGuarantees.`nonArb_collected`

$$\begin{gathered}
round \geq 1 \land transmitter \notin arbFaulty \Rightarrow \\
(\exists v \cdot collected = \texttt{MODULE} \times \{\{v\}\}) \lor collected = \texttt{MODULE} \times \{\varnothing\}
\end{gathered}$$

which ensures that a non-arb-faulty transmitter ensures that the range of the value tables of all modules is either the singleton set $\{v\}$ or the empty set. In the first case, *vote* must choose from this set, it has no choice but to select v. In the case of $\varnothing$ all value tables are identical and, hence, also any voting on them. ■

4 Byzantine Agreement Protocols

In the last years, a varienty of closely related variations of the byzantine agreement protocols have been published by different authors. This section will give a very brief classification of some of them, including those modelled in this report.

- [5] The original presentation came up with the basic protocols "Oral Messages" (OM) and "Signed Messages" (SM).
- [10] introduces the hybrid fault model in a protocol variant called Z.
- [7] combines the protocol of oral messages OM with the hybrid fault model to the protocol "Oral Messages, Hybrid" (OMH).
- [3] by authors of the same group describes more combinations of protocols:
  - "Signed Messages, Hybrid" (SMH)
  - "OMH with Authentication" (OMHA)
  - "Z with Authentication" (ZA)

We compare these protocols in terms of the following properties:

S: Signatures – messages are signed
A: Authentication – authentication is sound, i. e. signatures are reliable (the usual assumption when using signatures)
H: Hybrid fault model – distinguishing between manifest, symmetric and arbitrary faults
M: Messages reduced – optimised number of messages
V: Voting – majority voting
R: Reporting errors – reporting errors using a report (or wrapper) function R, distinguishing between E and $R(E)$

Please see table 1 for the overview.

Protocol	S	A	H	M	V	R
OM	.	.	.	.	V	.[c]
OMH	.	.	H	.	V	R
OMHA	S	?	H	.	V	R
SM	S	A	.	M	.	.[b]
SMH	S	A	H	M	.	.
Z	.	.	H	.	V	.
ZA	S	.[a]	H	.	V	.

S: Signatures
A: Authentication (reliable signatures)
H: Hybrid fault model
M: Messages reduced
V: Voting
R: Reporting errors

[a] The main idea behind ZA.
[b] No explicit E value, but implicit by missing messages.
[c] No explicit E value, missing messages are treated as a default value ("RETREAT").

Table 1: Overview of byzantine agreement protocols

5 Definitorial Extensions

The refinement structure presented Sect. 2 introduced refinements capturing technical details of the agreement proofs. We later claimed that certain invariants in these refinements are also invariants of the refined machines. That is of course not the case in general but holds here because the technical refinements are of a particularly simple kind.

Intuitively, a definitorial extension only *adds deterministic behaviour* without touching existing model properties. Any trace of a definitorial extension can therefore be easily projected to a trace of the refined machine. An invariant of the extension which is syntactically valid in both machines is therefore also an invariant of the original machine.

Definition (Definitorial Extension) *A machine N is called a* definitorial extension *of a machine M if:*

1. *M and N see the same contexts.*
2. *$v_M \subseteq v_N$, i.e., N extends[10] the set of variables of M.*
3. *For every event e^M in M there is exactly one event e^N by the same name in N such that e^N and e^M coincide in the sets of parameter variables and guards. The actions in e^N comprise all actions of e^M and, additionally, deterministic actions $x := E^e_x(v_N)$ for the new variables $x \in v_N \setminus v_M$.*
4. *No other events are defined in N.*

Proposition *If an Event-B Machine N is a definitorial extension of M, then any invariant $I(v_M)$ in N which syntactically uses only variables in v_M is also an invariant for M.*

Proof Due to the nature of the definitorial extension, the before-after-predicates of an event e have the following relationship (with $v_N \setminus v_M = \{x_1, \ldots, x_n\}$)

$$BAP^N_e(v_N, v'_N) \Leftrightarrow BAP^M_e(v_M, v'_M) \wedge x'_1 = E^e_{x_1}(v_N) \wedge \ldots \wedge x'_n = E^e_{x_n}(v_N)$$

and for the initialisation

$$BAP^N_{init}(v'_N) \Leftrightarrow BAP^M_{init}(v'_M) \wedge x'_1 = E^{init}_{x_1} \wedge \ldots \wedge x'_n = E^{init}_{x_n} \ .$$

For every trace[11] $S^M := (s^M_0 \xrightarrow{e_o} s^M_1 \xrightarrow{e_1} s^M_2 \xrightarrow{e_2} \ldots)$ of machine M we can extend the states[12] s^M_i to states s^N_i for machine N. We specify $s^N_0(x) := val(E^{init}_x)$ (expressions in the initialising event must

[10] Variable sets of machines are disjoint by definition. Thus, to be precise we should say there is a variable set $\bar{v}_M$ with $\bar{v}_M \subseteq v_N$ for which $\bar{v}_M = v_M$ is implied by the glueing invariant. For simplicity, we identify v_M with the copy $\bar{v}_M$ here.

[11] A trace for a machine is a sequence $(s_i)_{i\in\mathbb{N}}$ of states, s.t. s_0 is a state the initialising event can result in and (s_i, s_{i+1}) is in the transition relation of an event e of the machine. We notate the used event e above the arrow we write between states of a trace

[12] A state of a machine is a function that assigns to every variable a value of the domain of the variable's type.

not depend on variables) and $s_{i+1}^N(x) := val_{s_i^N}(E_x^{e_i}(v_N))$ for all $x \in v_N \setminus v_M$. The resulting trace $S^N := (s_0^N \xrightarrow{e_0} s_1^N \xrightarrow{e_1} s_2^N \to \ldots)$ satisfies all necessary before-after-predicates for N by construction since S^M satisfies the according predicates for M. Now, if $I(v_M)$ is an invariant for N in which only variables in v_M are used, the valuations $val_{s_i^M}(I(v_M)) = val_{s_i^N}(I(v_M))$ are equal for all i as s_i^M and s_i^N coincide on v_M. Being an invariant, $I(v_M)$ is always true in all states of S^N, and, hence, also in all states of S^M. ∎

All technical machines (i.e. *TECH) are definitorial extensions of their non-technical counterparts which implies that the corresponding agreement invariants can be lifted.

6 Lessons Learnt

6.1 Event-B as modelling language

Unlike in more general purpose formal systems (like PVS, Isabelle, Coq), Event-B imposes a rather strict corset on the means one can use to model systems. One thing is the commitment to (first-order) set-theory as the underlying logic. Another is the requirement to model state transitions as events using generalised substitutions. The introduction of events allows a natural notion of refinement over them.

We will now describe how, with respect to these aspects, the Event-B method was suited for the task of formalising the byzantine agreement problem.

Using Sets We believe that for the present problem, sets and relations were good means to express the necessary structures and their alterations. Event-B is evidently built with the primary focus on *binary* relations, higher arity is more complicated to model, as built-in operators (such as $\lessdot$, $\lhd$, ...) cannot be applied as conveniently. There is a wide variety of set-theoretic operators which allow to express even complex issues rather concisely—once one is used to them. They have, however, in comparison to the more verbose ausfhrungen in plain first order logic the disadvantage that they appear cryptic to an outsider not used to the symbols of B/Event-B.

At one point, we wanted to slightly leave the purely set-oriented view and introduced ordered sequences. We tried both a formalisation as abstract data types (using two constructors *cons* and *nil*) and as partial functions from an initial interval of the natural numbers (see Sect. 2.9). This could not be done in a generic way since Event-B does not support general parametrised types (like "list of X" for some type X), but only for one particular type of list elements. Also, the tool support for this additional data structure was poor.

Using Events The Byzantine Agreement belongs to a family of algorithms which, after an initialising step, repeatedly perform the same operations again and again. It also is a protocol which allows to easily divide the progress of the algorithm into steps or rounds.

These two properties make the procedure suitable for modelling it using events. In this particular case there is, apart from the initialisation, only one event Round which describes the effects of receiving and sending messages during one round of the protocol.

If the protocol underwent several phases, we would probably need to add an artificial variable which would encode the phase we would be in. If one step was so complex that it could not be adequately expressed by a generalised substitution but rather by something like a loop, we would then have to simulate that loop by a series of events, and would have to add to the state description to inhibit other events in the meantime.

Refinement We were able to subdivide the modelling process into eleven steps of refinement of different natures. Some refinements enriched the model by new aspects of the protocol, some performed a representation change of the machine state, mainly by changing the way in which a single message was represented. The mechanism certainly helped to structure the model.

6.2 The tool Rodin

The tool Rodin was used to discharge the arisen proof obligations for our model. The handling and the user interface are very convenient since the tool is designed as a modification of the eclipse environment.

In the beginning of our project, the tool seemed astonishingly stable and never crashed. With growing sizes of terms and sequents, sudden crashes, strange behaviour and out-of-memory-exceptions became more frequent[13].

The interactive prover mechanism, though being quite nice and intuitive to use, lacks strength in general. Only few rules can be applied manually, one cannot quite prove interactively without using the external provers. In particular, the interactive prover lacked important rewrite simplification rules for set comprehensions and λ-abstractions. Fortunately, these rules were added in Rel. 1.1.

The automatic provers that are shipped with Rodin have their strengths and weaknesses:

- Arithmetically, only one (ML) prover supports some linear arithmetic. Its power is very limited as it apparently expects the input to be of a a certain format. For instance, at one point, when $x + 1 \geq 0$ was on the sequent (for $x \in \mathbb{N}$), we had to add an additional hypothesis $x \geq -1$ which then allowed the procedure to finish the proof automatically.
- Quantification instantiation is not a strength of any of the proof engines. Only in very few and very obvious cases (like $(\forall x \cdot p(x)) \rightarrow p(t)$), necessary instantiations were done automatically and successfully, apparently also only for a single variable. When instantiating lemmata, we always had to give terms for the universally quantified variables. At the same time, we must admit that the lemmata themselves were quite complicated properties.
- Set theoretic constructs. The lemmata built into the ML prover seem to be rather extensive. At many points, the provers (in particular `newPP`) surprised by closing not quite so obvious goals automatically.

  On other occasions, the proof would not close unless we manually performed a very basic step manually (such as `imp_right` e.g.).
- Cardinalities seem a step child of the provers. Fortunately, the interactive component compensates by establishing a couple of deduction rules. Unreasonably often, we had to add the obvious hypothesis $\text{card}(t) \geq 0$ for some term t.

In general, quite many case distinctions and added hypotheses were needed to guide the automatic procedures. After a while, one gained a feeling on what to do as the decision points where symptomatic.

The automatic decision procedures act as black boxes which do not give any justification for their closing a branch; they are comparable to modern SMT procedures in that point. The procedures not only operate on the formulas on the current sequent but also possibly incorporate formulas from the list of available valid hypotheses so that it was not possible to decide which lemmata were actually used during the proof of a property. When selecting lemmata manually ("search hypotheses"), the names are not listed (although all invariants and axioms have names). Equally, the names do not appear in the proof tree, which would be a great help for retracing proofs.

If the user adds and axiomatises new data structures, they may find that the tool is not as supporting as they would wish. When we modelled sequences in 2.9, we were obliged to prove many times the fact that the concatenation of a value to a list is a list again. A mechanism which allows the user to define inference rules which then could be applied manually (or even automatically) could definitely reduce proof efforts under certain circumstances. The authors are glad that such an extension mechanism is on its way and will be available in future versions.

7 Conclusion

In this report we describe our formal models of byzantine agreement protocols. In particular, we model the known protocols ZA and the SM using the Event-B method. Our model comprises 4 contexts and 12 machines, with a total of 106 invariants. We used the tool Rodin to discharge all of the 322 proof obligation for the model. 74 of them could could be closed automatically, while more than 75% had to be proven manually, some with a considerable amount of interaction effort. Approximately three man-months were invested in the design, implementation and verification of the model.

Despite the fact that a protocol has been proven correct, it is still fairly easy to come up with an implementation which only seemingly implements this protocol but deviates in such a way that guarantees no longer hold. It should be subject of a further investigation to extend the refinement

[13] Astonishingly, Rel. 1.0 was considerably more stable than Rel. 1.1 and Rel. 1.2 in our case.

chain given in this report more towards a real implementation of an algorithm and to establish a refinement relation between model and implementation.

References

[1] Jean-Raymond Abrial. *Modeling in Event-B.* Cambridge Univ. Press, 2010.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In Zhiming Liu and Jifeng He, editors, *ICFEM 2006*, volume Lectur. Springer, June 2006.

[3] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In Ravishankar K. Iyer, Michele Morganti, W. Kent Fuchs, and Virgil Gligor, editors, *Dependable Computing for Critical Applications—5*, volume 10 of *Dependable Computing and Fault Tolerant Systems*, pages 139–157, Champaign, IL, Sept. 1995. IEEE Computer Society.

[4] Stefan Hallerstede. Incremental system modelling in Event-B. In *FMCO 2008*, LNCS, pages 139–158. Springer-Verlag, 2008.

[5] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[6] Michael Leuschel and Michael Butler. Prob: an automated analysis toolset for the b method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, 2008.

[7] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Fault-Tolerant Computing Symposium, FTCS 23*, pages 402–411, Toulouse, France, June 1993. IEEE Computer Society.

[8] Christophe Métayer. AnimB - B model animator. `http://www.animb.org/index.xml` (Accessed: 7 Apr 2010).

[9] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. of the ACM*, 27(2):228–234, 1980.

[10] Philip M. Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *SRDS*, pages 93–100, Columbus, Ohio, USA, October 1988. IEEE Computer Society Press.