Assurance of Distributed Algorithms and Systems: Runtime Checking of Safety and Liveness

Yanhong A. Liu(✉) and Scott D. Stoller

Computer Science Department, Stony Brook University, Stony Brook, NY, USA
{liu,stoller}@cs.stonybrook.edu

Abstract. This paper presents a general framework and methods for complete programming and checking of distributed algorithms at a high-level, as in pseudocode languages, but precisely specified and directly executable, as in formal specification languages and practical programming languages, respectively. The checking framework, as well as the writing of distributed algorithms and specification of their safety and liveness properties, use DistAlgo, a high-level language for distributed algorithms. We give a complete executable specification of the checking framework, with a complete example algorithm and example safety and liveness properties.

1 Introduction

Distributed systems are increasingly important in our increasingly connected world. Whether for distributed control and coordination or distributed data storage and processing, at the core are distributed algorithms.

It is well known that distributed algorithms are difficult to understand. That has led to significant effort in specifying these algorithms and verifying their properties, e.g., [5,13,36], as well as in developing specification languages and verification tools, e.g., TLA and TLA+ Toolbox [18,20,34], I/O Automata [31], and Ivy [33]. However, challenges remain for automated verification of practical distributed algorithms using theorem proving or model checking techniques, due to exorbitant manual effort and expertise required or prohibitive state-space explosion.

Runtime verification allows information to be extracted from a running system and used to check whether observed system behaviors satisfy certain properties, and to react based on the results of checking. It is the most effective complement to theorem proving and model checking for sophisticated algorithms and implementations. For routinely checking real distributed applications written in general-purpose programming languages, it is so far the only feasible practical solution.

This work was supported in part by NSF under grants CCF-1414078, CCF-1954837, CNS-1421893, and IIS-1447549, and ONR under grant N00014-20-1-2751.

J. Deshmukh and D. Ničković (Eds.): RV 2020, LNCS 12399, pp. 47–66, 2020.
https://doi.org/10.1007/978-3-030-60508-7_3

Many methods and related issues for doing such runtime checking have been studied, as discussed in Sect. 8. Such checking and all kinds of variations have also long been used extensively in practical program development, testing, debugging, and simulation for distributed algorithms. However, these studies and uses of runtime checking are either more abstract methods, not presented as executable programs, or involving significant programming using commonly-used programming languages, too large to present in complete and exact forms on paper.

This paper presents a general framework and methods for complete programming and checking of distributed algorithms at a high-level, as in pseudocode languages, but precisely specified and directly executable, as in formal specification languages and practical programming languages, respectively. The checking framework, as well as the writing of distributed algorithms and specification of their safety and liveness properties, use DistAlgo, a high-level language for distributed algorithms [21,29]. We give a complete executable specification of the checking framework, with a complete example algorithm and example safety and liveness properties.

The framework can check any desired properties against observed system behavior. Note that since any execution of a practical system is finite, the liveness properties we check are bounded liveness, that is, the desired properties hold within specified time bounds. The framework requires no change to the algorithm code to be checked. It puts the algorithm code, property specification, as well as fault simulation together with small configurations, thanks to the power of the DistAlgo language and compiler. The complete checking program then automatically intercepts messages sent and received by the distributed processes to be checked, with both logical and real times, and checks the specified properties at desired points as written.

This framework has been used in implementing, testing, debugging, simulation, and analysis of many well-known distributed algorithms, and in teaching. Our experiences included discovering improvements to both correctness and efficiency of some well-known algorithms, e.g., [23,25,28].

2 Distributed Algorithms and Their Safety and Liveness

Distributed algorithms are algorithms that run in distributed systems. Understanding and verifying their properties are central challenges for distributed computing.

Distributed Systems and Distributed Algorithms. A distributed system is a set of distributed processes. Each process has its own private memory that only it can access. Processes execute concurrently and communicate with each other by sending and receiving messages.

Distributed processes and communications are prone to various kinds of failures, depending on the underlying infrastructures. Processes may be slow, may crash, may later recover, and may even behave arbitrarily. Messages may be lost, delayed, duplicated, reordered, and even be arbitrarily changed.

Distributed algorithms are for solving problems that involve coordination, negotiation, etc. among distributed processes in the presence of possible failures. Due to nondeterminism from concurrency and uncertainty from failures, distributed algorithms are notoriously difficult to design, understand, and verify. Even as an algorithm executes in a distributed system, the state of the system is not fully observable, and the order of events cannot be fully determined. This led to Lamport's creation of logical clocks, which are fundamental in distributed systems [17].

Distributed computing problems are of an extremely wide variety, and a great number of distributed algorithms have been studied. e.g., [7,9,31]. Well-known problems range from distributed clock synchronization to distributed snapshot, from leader election to distributed mutual exclusion, from atomic commit to distributed consensus, and many more. We give two examples here:

– Distributed mutual exclusion. Distributed mutual exclusion is for multiple processes to access a shared resource mutually exclusively, in what is called a critical section, i.e., there can be at most one process in a critical section at a time.
  It is one of the most studied problems, e.g., [15,37], with at least dozens if not hundreds or more of proposed algorithms and variants. For example, Lamport's algorithm [17], introduced to show the use of logical clocks, was designed to guarantee that access to the resource is granted in the order of logical clock values of the requests.
– Distributed consensus. Distributed consensus is for a set of processes to agree on a single value or a continuing sequence of values, called single-value consensus or multi-value consensus, respectively.
  It is essential in any important service that maintains a state, including services provided by companies like Google and Amazon. This is because such services must use replication to tolerate failures caused by machine crashes, network outages, etc. Replicated processes must agree on the state of the service or the sequence of operations that have been performed, e.g., that a customer order has been placed and paid but not yet shipped, so that when some processes become unavailable, the remaining processes can continue to provide the service correctly.
  Even well-known algorithms and variants number at least dozens, starting from virtual synchrony [1–3], viewstamped replication [22,35], and Paxos [19].

These problems are at the core of distributed file systems, distributed databases, and fault-tolerant distributed services in general. New algorithms and variants for them are developed constantly, not to mention a vast number of other distributed algorithms, such as network protocols, distributed graph algorithms, and security protocols.

Safety and Liveness. Lamport [16] first formulated two types of properties of a distributed system: safety and liveness. Informally, a safety property states that some bad things will not happen, and a liveness property states that some good things will happen. We continue the two examples discussed earlier:

- For distributed mutual exclusion, a most important safety property is that at most one process is in a critical session at a time. A liveness property is that some requests are eventually served, and a stronger liveness property is that all requests are eventually served.
  For example, Lamport's algorithm [17] is designed to guarantee all these, and in fact, as mentioned above, to guarantee a stronger property—that all requests are served in the order of logical clock values. This stronger property can be interpreted and formulated as either a safety property, to mean that no requests are served out of the order of logical clock values, or a liveness property, to include that all requests are eventually served.
- For distributed consensus, there are two important safety properties: (1) agreement on the decided single value, in single-value consensus, or on the sequence of values, in multi-value consensus, by nonfaulty processes, and (2) validity of the decided value or values to be among allowed values. A liveness property for single-value consensus is that nonfaulty processes eventually decide on a value. A liveness property for multi-value consensus is that nonfaulty processes repeatedly decide on additional values in the sequence.
  Good distributed consensus algorithms, such as Paxos [19], guarantee the two safety properties, but they cannot guarantee the liveness property due to the well-known impossibility of consensus in asynchronous distributed systems even with only one faulty process subject to crash failures [6].

Specifying safety and liveness properties is nontrivial, especially liveness properties, even informally. For example, liveness for many consensus algorithms and variants has been left unspecified, or specified too weakly to be useful or too strongly to be possible [4].

Safety and liveness are, in general, global properties about multiple processes. Checking them requires knowing the states of multiple processes. However, the state of a process is private to that process and cannot be accessed directly by any other process. The best one can do is to observe a process by intercepting messages sent and received by that process, and determine the state of the system and desired properties conservatively or approximately, and with a delay.

We use checker to refer to a process that observes the sending and receiving of messages by a set of processes and checks desired properties.

3 A Powerful Language for Distributed Programming

A powerful language for distributed programming must allow (1) easy creation of distributed processes and communication channels for sending messages, (2) easy handling of received messages, both synchronously (with waiting) and asynchronously (without waiting), (3) easy processing of all information communicated as well as a process's own data, and (4) easy configuration of basic elements for real execution on distributed machines.

Running Example: The Polling Problem. We introduce a simple but essential problem, which we call the polling problem, as a running example:

> A poller process sends a question to a set of pollee processes, waits to receive answers to the question from all of them, and then sends an outcome message to them.

Small variations of this problem include waiting to receive replies from a subset of the pollee processes, such as a majority or a quorum, instead of all of them.

This problem is essential because any process working with a set of other processes requires talking to and hearing back from those processes one way or another. This problem manifests widely in well-known distributed algorithms, including algorithms for distributed mutual exclusion, atomic commit, and distributed consensus. This problem also manifests itself in everyday life, such as an instructor giving assignments to students, a chairperson soliciting suggestions from committee members, or a campaign organizer sending a poll to voters.

The problem appears simple, but is nontrivial, even without process failures or message delays or losses, because processes are generally communicating with multiple processes and doing other things at the same time. Consider some examples:

- When the poller receives a message from a pollee, how does the poller know it is replying to a particular question? The pollee might happen to send something to the poller with the same format as an expected reply, and send it shortly after the question was sent.
- How does the poller know it has received replies from all pollees? It could compare the number of replies to the number of pollees, but a pollee might send multiple replies, or a communication channel might duplicate messages.

The problem becomes even harder if processes can fail and messages may be lost or delayed. It becomes even more challenging if processes can fake identities and messages can be altered or counterfeited. In the latter cases, processes need to use security protocols for authentication and secure communication. Although we do not consider those problems further in this tutorial, we have extended DistAlgo with a high-level cryptographic API for expressing such security protocols [14].

Figure 1 shows a complete polling program written in DistAlgo. Process classes `P` and `R` specify the poller and responder (i.e., pollee) processes, respectively. Definitions `run` and `receive` specify the main algorithm. The core of the algorithm is on lines 4–6, 8, 13, and 15–16. The rest puts all together, plus setting up processes, starting them, and outputting about replies and outcomes of the polling. The details are explained in examples as we describe next the DistAlgo language used.

DistAlgo, a Language for Distributed Algorithms. DistAlgo supports easy distributed programming by building on an object-oriented programming language, with a formal operational semantics [29], and with an open-source implementation [21] that extends Python.

Because the implementation uses the Python parser, it uses Python syntax such as `send(m, to=p)` instead of the ideal `send m to p` for sending message `m` to process `p`. For the same reason, it uses `from_` in place of the ideal `from` because

```
 1 class P (process):             # define poller process class
 2   def setup(rs): pass          # take a set of responder processes
 3   def run():
 4     self.t = logical_clock()  # get logical clock time as question id
 5     send((’question’, question(), t), to= rs)
 6     await each(r in rs, has= some(received((’reply’,_,_t), from_=_r)))
 7     output(’-- received Y from:’, setof(r, received((’reply’,’Y’,_t), from_= r)))
 8     send((’outcome’, outcome()), to= rs)

 9   def question(): return ’Did you attend RV (Y/N)?’ # create question
10   def outcome(): return countof(r, received((’reply’,’Y’,_t), from_= r))

11 class R (process):             # define responder (i.e., pollee) process class
12   def run():
13     await some(received((’outcome’, o)))
14     output(’== received outcome:’, o)
15   def receive(msg=(’question’, q, t), from_= p):
16     send((’reply’, reply(q), t), to= p)

17   def reply(q): import random; return random.choice([’Y’,’N’]) # create reply

18 def main():
19   config(clock = lamport)      # use Lamport clock
20   config(channel = reliable)  # use reliable channels
21   rs = new(R, [], num= 10)     # create and set up 10 responders
22   p = new(P, [rs])             # create and set up poller process p
23   start(rs)                    # start responders
24   start(p)                     # start poller
25
```

Fig. 1. Polling program in DistAlgo.

the latter is a keyword in Python. A final quirk is that we indicate a previously bound variable in patterns with prefix _ in place of the ideal `=` because _ is the only symbol allowed besides letters and numbers in identifiers. Besides the language constructs explained, commonly used constructs in Python are used, for no operation (`pass`), assignments (v `=` e), etc.

Distributed Processes That Can Send Messages. A type P of distributed processes is defined by `class` P `(process):` *body*, e.g., lines 1–10 in Fig. 1. The body may contain

– a `setup` definition for taking in and setting up the values used by a type P process, e.g., line 2,
– a `run` definition for running the main control flow of the process, e.g., lines 3–8,
– other helper definitions, e.g., lines 9–10, and
– `receive` definitions for handling received messages, e.g., lines 15–16.

A process can refer to itself as `self`. Expression `self.`*attr* (or *attr* when there is no ambiguity) refers to the value of *attr* in the process. *ps* `= new(`P`,`*args*`,`*num*`)` creates *num* (default to 1) new processes of type P, optionally passing in the values of *args* to `setup`, and assigns the set of new processes to *ps*, e.g., lines 21 and 22. `start(`*ps*`)` starts `run()` of processes *ps*, e.g., lines 23 and 24. A separate `setup(`*ps*`,`*args*`)` can also set up processes *ps* with the values of *args*.

Processes can send messages: `send(`m`, to=`*ps*`)` sends message m to processes *ps*, e.g., line 5.

Control Flow for Handling Received Messages. Received messages can be handled both asynchronously, using `receive` definitions, and synchronously, using `await` statements.

- A `receive` definition, `def receive (msg=`m`, from_=`p`)`, handles, at yield points, un-handled messages that match m from p, e.g., lines 15–16. There is a yield point before each `await` statement, e.g., line 6, for handling messages while waiting. The `from_` clause is optional.
- An `await` statement, `await` *cond*, waits for *cond* to be true, e.g., line 6. A more general statement, `if await` $cond_1$`:` $stmt_1$ `elif` ... `elif` $cond_k$`:` $stmt_k$ `elif timeout(`t`):` *stmt*, waits for one of $cond_1$, ..., $cond_k$ to be true or a timeout after t seconds, and then nondeterministically selects one of $stmt_1$, ..., $stmt_k$, *stmt* whose conditions are true to execute.

High-Level Queries for Synchronization Conditions. High-level queries can be used over message histories, and patterns can be used for matching messages.

- Histories of messages sent and received by a process are automatically kept in variables `sent` and `received`, respectively. `sent` is updated at each `send` statement, by adding each message sent. `received` is updated at yield points, by adding un-handled messages before executing all matching `receive` definitions.

  Expression `sent(`m`, to=`p`)` is equivalent to `(`m`,`p`)` in sent. It returns true iff a message that matches `(`m`,`p`)` is in `sent`. The `to` clause is optional. `received(`m`, from_=`p`)` is similar.
- A pattern can be used to match a message, in `sent` and `received`, and by a `receive` definition. A constant value, such as `'respond'`, or a previously bound variable, indicated with prefix _, in the pattern must match the corresponding components of the message. An underscore _ by itself matches anything. Previously unbound variables in the pattern are bound to the corresponding components in the matched message.

  For example, `received(('reply','Y',_t), from_=r)` on line 7 matches in `received` every message that is a 3-tuple with components `'reply'`, `'Y'`, and the value of `t`, and binds `r` to the sender.

A query can be a comprehension, aggregation, or quantification over sets or sequences.

- A comprehension, `setof(`e`,` v_1 `in` s_1`, ...,` v_k `in` s_k`,` *cond*`)`, where each v_i can be a pattern, returns the set of values of e for all combinations of values of variables that satisfy all v_i `in` s_i clauses and satisfy condition *cond*, e.g., the comprehension on line 7.
- An aggregation, similar to a comprehension but with an aggregation operator such as `countof` or `maxof`, returns the value of applying the aggregation operator to the collection argument, e.g., the `countof` query on line 10.

- A universal quantification, `each(`v_1 `in` s_1`, ...,` v_k `in` s_k `has=`*cond*`)`, returns true iff, for all combinations of values of variables that satisfy all v_i `in` s_i clauses, *cond* holds, e.g., the `each` query on line 6.
- An existential quantification, `some(`v_1 `in` s_1`, ...,` v_k `in` s_k `has=`*cond*`)`, returns true iff, for some combinations of values of variables that satisfy all v_i `in` s_i clauses, *cond* holds, e.g., the `some` query on line 13. When the query returns true, all variables in the query are bound to a combination of satisfying values, called a witness, e.g., `o` on line 13.

Configuration for Setting Up and Running. Configuration for requirements such as use of logical clocks and reliable channels can be specified in a `main` definition, e.g., lines 19–20. When Lamport's logical clocks are used, DistAlgo configures sending and receiving of a message to update the clock value, and defines a function `logical_clock()` that returns the clock value. Processes can then be created, setup, and started. In general, `new` can have an additional argument, specifying remote nodes where the newly created processes will run; the default is the local node.

4 Formal Specification of Safety and Liveness

When specifying properties about multiple distributed processes, we refer to the `sent` and `received` of a process *p* as *p*`.sent` and *p*`.received`. We will use ideal syntax in this section in presenting the safety and liveness properties, e.g., `p.received m from p at t` instead of `p.received(m, from_=p, clk=t)`.

Specifying Safety. Despite being a small and seemingly simple example, a wide variety of safety properties can be desired for polling. We consider two of them:

(S1) *The poller has received a reply to the question from each pollee when sending the outcome.*
This property does not require checking multiple distributed processes, because it uses information about only one process, the poller. In fact, in the program in Fig. 1, it is easy to see that this property is implemented clearly in the poller's run method.
We use this as an example for three reasons: (1) it allows the reader to contrast how this is specified and checked by the checker compared with by the poller, (2) such checks can be important when we do not have access to the internals of a process but can observe messages sent to and from the process, and (3) even if we have access to the internals, it may be unclear whether the implementation ensures the desired property and thus we still need to check it.

(S2) *Each pollee has received the same outcome when the program ends.*
This property requires checking multiple distributed processes, because the needed information is not available at a single process.
We use this example to show such properties can be specified and checked easily at the checker, conservatively ensuring safety despite the general impossibility results due to message delays, etc.

Consider property (S1). The checker will be informed about all processes and all messages sent and received by each process. Also, it can use universal and existential quantifications, just as in line 6 of the poller's code in Fig. 1. However, there are two issues:

1) How does the checker know "the" question? Inside the poller, "the" question is identified by the timestamp in variable `t`, which is used in the subsequent tests of the replies. To check from outside, the checker needs to observe the question and its id first, yielding a partial specification for (S1):

   ```
   some p.sent ('question', _, t) has
    each r in rs has some =p.received ('reply', _, =t) from =r
   ```

   If one knows that the poller sent only one question, then the `some` above binds exactly that question. Otherwise, one could easily check this by adding a conjunct `count {t: p.sent ('question', _, t)} == 1`.
2) How does the checker know that all replies had been received when the outcome was sent (Note that a similar question about identifying "the" outcome can be handled the same way as for "the" question.) Inside the poller, it is easy to see that tests of the replies occur before the sending of the `'outcome'` message. Outside the poller, we cannot count on the order that the checker receives messages to determine the order of events. The checker needs to use the timestamps from the logical clocks.

   ```
   some p.sent ('question', _, t), p.sent ('outcome', _) at t1 has
    each r in rs has some =p.received('reply',_,=t) from =r at t2 has t1>t2
   ```

   Note the added `p.sent ('outcome', _) at t1` on the first line and `at t2 has t1 > t2` on the second line.

Note that when the receiver or logical time of a sent message is not used, it is omitted from the property specification; it could also be included and matched with an underscore, e.g., `p.sent m to _ at _`.

Consider property (S2), which is now easy, using the same idea to identify "the" outcome `o` based on the outcome message sent by the poller:

```
some p.sent ('outcome', o) has
  each r in rs has some =r.received ('outcome', =o)
```

The checker just needs to check this at the end.

Specifying Liveness. Specifying liveness requires language features not used in the algorithm description. We use the same specification language we introduced earlier [4]. In particular,

`evt` *cond*

where `evt` is read as "eventually", denotes that *cond* holds at some time in the duration that starts from the time under discussion, i.e., eventually, *cond* holds.

Many different liveness properties can be desired. We consider two of them:

(L1) *The poller eventually receives a reply to the question.*
This assumes that a question was sent and covers the duration from then to receiving the first reply.
We use this example because it is the first indication to the poller that the polling really started. We also use receiving the first reply to show a small variation from receiving all replies.

(L2) *Eventually each pollee receives the outcome.*
This assumes that an outcome was sent and covers the entire duration of the polling.
We use this example because it expresses the completion of the entire algorithm.

For (L1), one can simply specify it as an `evt` followed by the partial specification for (S1) except with `each r in rs` replaced with `some r in rs`:

```
evt some p.sent ('question', _, t) has
   some r in rs has some =p.received ('reply', _, =t) from =r
```

In practice, one always estimates an upper bound for message passing time and poll filling time. So one can calculate an upper bound on the expected time from sending the question to receiving the first reply, and be alerted by a timeout if this bound is not met.

For (L2), one can see that this just needs an `evt` before the property specified for (S2):

```
evt some p.sent ('outcome', o) has
       each r in rs has some =r.received ('outcome', =o)
```

In practical terms, (L2) means that the program terminates and (S2) holds when the program ends. Thus, with (S2) checked as a safety property, (L2) boils down to checking that the program terminates.

Conceptually, `evt` properties are checked against infinite executions. In practice, they are checked against finite executions by imposing a bound on when the property should hold, and reporting a violation if the property does not hold by then. From a formal perspective, imposing this time bound changes the liveness property to a safety property.

5 Checking Safety

We describe a general framework for checking safety through observation by a checker external to all original processes in the system. The checker observes all processes and the messages they send and receive. We then discuss variations and optimizations.

Extending Original Processes to Be Checked. The basic idea is: each process `p`, when sending or receiving a message, sends information about the sending or receiving to the checker. The checker uses this information to check properties of the original processes.

The information sent to the checker may include (1) whether the message is being sent or received by `p`, indicated by `'sent'` and `'rcvd'`, respectively, (2) the message content, (3) the receiver or receivers (for a message sent) and sender (for a message received), and (4) the logical timestamp of the sending or receiving, if a logical clock is used. In general, it may include any subset of these, or add any other information that is available and useful for checking properties of interest.

With ideal channels to send such information to the checker, the checker can extract all the information using the following correspondence:

```
p.sent m to qs at t      ⟺ checker received('sent' m to qs at t) from p
p.received m from q at t ⟺ checker received('rcvd' m from q at t) from p
```

Sending the information can be done by extending the original processes, so the original program is unchanged. The extended processes just need to (1) extend the `send` operation to send information about the sending to the checker, and (2) add a `receive` handler for all messages received to send information about the receiving to the checker. A checker process can then specify the safety conditions and check them any time it desires; to check at the end, it needs to specify a condition to detect the end.

Figure 2 shows safety checking for the polling example. It imports the original program `polling.da` as a module, and extends processes `P` and `R` to take `checker` as an argument at setup. In extended `P`, it extends `send` and adds `receive` to send all 4 kinds of information listed to `checker` (lines 4–8). In extended `R`, it sends 3 kinds of information, omitting logical times (lines 12–16). It then defines process `Checker` that takes in `p` and `rs` at setup, waits for a condition to detect the end of the polling (line 20), and checks safety properties (S1) and (S2) (lines 22–31). The `main` method is the same as in Fig. 1 except for the new and updated lines for adding the checker process, as noted in the comments.

Variations and Optimizations. When checking systems with many processes, a single checker process would be a bottleneck. The single checker framework can easily be extended to use a hierarchy of checkers, in which each checker observes a subset of original processes and/or child checkers and reports to a parent checker.

As an optimization, information not needed for the properties being checked can be omitted from messages sent to the checker, leading to more efficient executions and simpler patterns in specifying the conditions to be checked. In Fig. 2, process `R` already omits logical times from all messages to the checker. More information can be omitted. For example, process `P` can omit target processes, the 3rd component of the message, in information about sending. Additionally, process `R` can omit all information about sending and all but the second component about receiving. Furthermore, more refined patterns can be used when extending `send` to omit unused parts inside the message content, the second component. For example, the specific question in `'question'` messages can be omitted.

Instead of extending the original processes to be checked, an alternative is to let the original processes extend a checker process. While the former approach

```
import polling  # import the module, polling.da, to be checked

class P (process, polling.P):  # extend process P in module polling
  def setup(checker,rs): super().setup(rs)  # set up checker and original rs
  def send(m, to):               # override send to send information to checker
    super().send(m, to)          # do original send
    super().send(('sent', m, to, logical_clock()), to= checker) # send to checker
  def receive(msg= m, from_= fr):  # add a receive matching any message received
    super().send(('rcvd', m, fr, logical_clock()), to= checker) # send to checker

class R (process, polling.R):  # as above but extend process R instead
  def setup(checker): super().setup()  # as above but set up only checker
  def send(m, to):               # as above but not send logical time to checker
    super().send(m, to)
    super().send(('sent', m, to), to= checker)
  def receive(msg= m, from_= fr):  # as above but not send logical time to checker
    super().send(('rcvd', m, fr), to= checker)

class Checker (process):
  def setup(p,rs): pass          # pass in processes p and rs
  def run():
    await each(r in rs, has= some(received(('rcvd', ('outcome',_), _), from_=_r)))
    output('~~ polling ended. checking safety:', S1(), S2())
  def S1():
    return some(received(('sent', ('question',_,t), _, _), from_=_p),
                received(('sent', ('outcome',_), _, t1), from_=_p), has=
                each(r in rs, has=
                     some(received(('rcvd', ('reply',_,_t), _r, t2),from_=_p),
                          has= t1 > t2)))
  def S2():
    return some(received(('sent', ('outcome',o), _, _), from_=_p), has=
                each(r in rs, has=
                     some(received(('rcvd', ('outcome',_o), _), from_=_r))))
def main():
  config(clock = lamport)
  config(channel = reliable)
  checker = new(Checker)           # create checker
  rs = new(R, [checker], num= 10)  # as in polling but add checker
  p = new(P, [checker, rs])        # as in polling but add checker
  setup(checker, [p,rs])           # setup checker with p and rs
  start(checker)                   # start checker
  start(rs)
  start(p)
```

Fig. 2. Checking safety for the polling program.

requires no change at all to the original processes, the latter approach requires small changes: (1) to each original process class, (a) add the checker process class as a base class and (b) add a `setup` parameter to pass in the checker process, and (2) in `main`, (a) create, setup, and start the checker process and (b) in the call to `new` or `setup` for each original process, add the checker process as an additional argument. The advantage of this alternative approach is that the same checker class can be used for checking different programs when the same checking is desired. An example use is for benchmarking the `run` method of different programs[1].

While checking safety using our framework is already relatively easy, higher-level declarative languages can be designed for specifying the desired checks, and specifications in such languages can be compiled into optimized checking programs that require no manual changes to the original programs.

[1] http://github.com/DistAlgo/distalgo/blob/master/benchmarks/controller.da.

6 Checking Liveness

As discussed in Sect. 4, in finite executions, checking liveness boils down to safety checking plus use of timeouts to check that properties hold within an expected amount of time. For the polling example, checking timeouts plus the same or similar conditions as (S1) and (S2) corresponds to what can be checked for (L1) and (L2). We describe a general framework for easily checking timeouts during program execution based on elapsed real time at the checker process. Using real time at the checker avoids assumptions about clock synchronization. We then discuss variations and optimizations.

Extending Original Processes to Be Checked. The same framework to extend original processes for safety checking can be used for liveness checking. One only needs to specify checks for timeouts instead of or in addition to safety checks. We show how timeouts between observations of any two sending or receiving events, as well as a timeout for the entire execution, can easily be checked, even with multiple timeout checks running concurrently.

Given any two sending or receiving events `e1` and `e2`, we check that after `e1` is observed by the checker, `e2` is observed within a specified time bound. There are two steps:

1) When the checker receives `e1`, it starts a timer for the specified time bound. Each timer runs in a separate thread and, when it times out, it sends a timeout message to the checker.
2) When the checker receives a timeout message, it checks whether the expected event `e2` has been observed. If yes, it does nothing. Otherwise, it reports a violation of the time bound requirement.

All time bounds are specified in a map, which maps a name for a pair of events to the required time bound from observing the first event until observing the second event.

This framework can easily be generalized to check conditions involving any number of events. When the timeout happens, instead of checking whether one specific event `e2` has been observed, the checker can check whether several expected events have all been observed, or whether any other desired condition on the set of observed events holds.

A time bound for the entire execution of the algorithm can be set and checked separately, in addition to checking any other safety and liveness properties, to directly check overall termination, using an appropriate condition to detect whether the algorithm has completed successfully.

Figure 3 shows timeout checking for the polling example. To check liveness instead of safety, one could use exactly the same program as for safety check except for the added `import`'s and `TIMEOUT` map at the top and a rewritten `Checker` process. To check timeouts in addition to safety, the `Checker` process in Fig. 3 can extend the `Checker` process in Fig. 2, and just add the function calls `super().S1()` and `super().S2()` at the end of the `run` method here.

Modules `threading` and `time` are imported in order to run each timer in a separate thread. A dictionary `TIMEOUTS` holds the map of time bounds (in seconds) for different pairs of events: `'q-r'` is for the poller sending the question and the poller receiving the first reply, corresponding to (L1), and `'q-o'` is for the poller sending the question and all pollees receiving the outcome, corresponding to (L2). The dictionary also includes an entry `'total'` with a time bound for the entire execution of the algorithm.

The `Checker` process waits for the same condition, as for safety checking, to detect the end of the polling (line 8–9), but with a timeout for `'total'` (line 10) while waiting. It starts two timers corresponding to (L1) and (L2) when observing the question was sent (lines 13–15), and checks and reports timeouts when any timer times out (lines 22–28).

```
 1 import threading
 2 import time
 3 TIMEOUTS = {'q-r':0.00001, 'q-o':0.0001, 'total':1}
 4                                       # map of timeout names to time bounds
 5 class Checker (process):
 6   def setup(p,rs): pass
 7   def run():
 8     if await each(r in rs, has=
 9                   some(received(('rcvd', ('outcome',_), _), from_=_r))): pass
10     elif timeout(TIMEOUTS['total']):  # entire execution times out
11       output('!! total timeout receiving outcome by all pollees')
12     output('~~ polling ended. timeouts checked')

13   def receive(msg=('sent', ('question',_,t), ps, _)):  # at 1st of two events
14     runtimer('q-r',t,ps)  # start timer 'q-r' that starts at this event
15     runtimer('q-o',t,ps)  # start timer 'q-o' that starts at this event
16
17   def runtimer(*args):   # run timer in a separate thread passing in all args
18     threading.Thread(target=timer, args=args).start()
19   def timer(name, *rest):  # timer, taking timer name and rest of arguments
20     time.sleep(TIMEOUTS[name])  # sleep the time bound for the timer name
21     send(('timeout', name, *rest), to= self)  # send timeout to chcker itself

22   def receive(msg=('timeout', 'q-r', t, ps)):  # at timeout for 'q-r'
23     if not some(received(('rcvd', ('reply',_,_t), _, _), from_=_)): # check 2nd event
24       output('!! L1 timeout receiving the first reply by the poller', t, ps)
25   def receive(msg=('timeout', 'q-o', t, ps)):  # at timoeout for 'q-o'
26     if not each(r in rs, has=  # check 2nd event
27                 some(received(('rcvd', ('outcome',_), _), from_=_r))):
28       output('!! L2 timeout receiving outcome by all pollees', t, ps, r)
```

Fig. 3. Checking timeouts for the polling program.

Variations and Optimizations. Variations and optimizations for checking safety can also be used for checking liveness. Checking timeouts using real time is the additional challenge.

In the program in Fig. 3, the timeout `'q-o'` for (L2) waits for the same condition as for detecting the end of polling in `run`, and covers almost the entire execution. Therefore, the test for detecting the end of polling in `run` is unnecessary in this case, and the entire body of `run` may simply be `await (timeout('total'))`. When a timeout `'q-o'` is received, the checker could terminate itself by importing `sys` and calling `sys.exit()`. Of course after either timeout for `'q-o'` or timeout in `run`, the checker process could also do anything else helpful instead of terminating itself.

Instead of or in addition using real time at the checker, one could use real time at the original processes. A clock synchronization algorithm can be used to improve the precision of clocks at different original processes and the overall precision and time bounds. Even without clock synchronization, using real time at the original processes can improve the precision and bounds for liveness properties involving multiple events at the same process, such as (L1).

Note that observing the start and end of an operation or entire program is also how performance measurements can be performed, as mentioned for benchmarking at the end of Sect. 5.

7 Implementation and Experiments

DistAlgo has been implemented as an extension of the Python programming language and used extensively in studying and teaching of distributed algorithms [29]. The framework discussed for checking safety and liveness properties has also been used extensively, in both ad hoc and more systematic ways. We describe using the DistAlgo implementation and our framework for execution and runtime checking.

Execution and Fault Simulation. DistAlgo is open source and available on github [21]. One can simply add it to the Python path after downloading, and run the `da` module in Python, e.g., running the program `polling.da` in Fig. 1 as `python -m da polling.da`.

For runtime checking, a checking program, such as the program in Fig. 2 can be run in the same way. More generally, implementations of three conceptual components are needed:

1) A distributed algorithm, plus input taken by the algorithm. This needs a complete executable program, such as `polling.da` in Fig. 1.
2) Safety and liveness requirements to be checked. These are expressed as executable functions that can be called at required points during the execution, such as functions `S1` and `S2` in Fig. 2 and the `receive` handlers in Fig. 3.
3) Process and communication failures to be considered. These correspond to executable configurations that can be executed for fault simulation, as described below.

Our framework puts these together naturally, with small configurations to observe processes and communications, with both logical and real times, thanks to the power of the DistAlgo language and compiler.

Fault simulation is essential for checking safety and liveness of complex algorithms in simulation of real distributed systems that are fault-prone. Both process and communication failures may happen, but the latter are much more frequent. Also, the former can be simulated with the latter, because a process interacts with other processes only through communication. For example, a crashed process is indistinguishable from one that stops sending messages to other processes.

With our existing framework for checking, we can simply use `send` to simulate all possible communication failures, including message

- loss: drop messages without sending;
- delay: delay messages before sending;
- duplication: send messages multiple times;
- reordering: delay sending messages until after sending later messages; and
- corruption for simulating Byzantine failures: change message before sending.

For example, to simulate a 1% chance of dropping a message sent by a `P` process, in the `send` method in Fig. 2, we can put `super().send(m,to)` inside a conditional:

```
if random.random() < 0.99: super().send(m,to)
```

and add `import random` before it.

Similarly, one may add fixed or random delays, duplications, reorderings, or a combination of them. A main issue to note is that, in general, one would want to send a delayed or duplicated message using a separate thread, to avoid delaying the execution of the algorithm.

Configuration options and values can be provided through command-line arguments and external files, as well as built-in language constructs. All these kinds have been provided in the DistAlgo language and compiler and used in DistAlgo programs. Similar mechanisms have been used in all kinds of system configurations for decades. A challenge is to design and implement a powerful, commonly accepted language for such configurations.

Experiments and Experience. For the running example, checking both safety and liveness, with the `Checker` process in Fig. 3 extending that in Fig. 2 but replacing the last line in `run` in Fig. 3 with the last line in `run` in Fig. 2 and adding `super().` before `S1()` and `S2()`, an example output is as shown below:

```
> python -m da .\polling_check_live.da
[54] da.api<MainProcess>:INFO: <Node_:75001> initialized at 127.0.0.1:(UdpTransport=37786, T
cpTransport=45837).
[54] da.api<MainProcess>:INFO: Starting program <module 'polling_check_live' from '.\\pollin
g_check_live.da'>...
[55] da.api<MainProcess>:INFO: Running iteration 1 ...
[56] da.api<MainProcess>:INFO: Waiting for remaining child processes to terminate...(Press "
Ctrl-Brk" to force kill)
[1446] da.api<MainProcess>:INFO: Main process terminated.
[160] polling_check.P<P:a900d>:OUTPUT: -- received Y from: {<R:a9007>, <R:a900b>, <R:a900c>,
 <R:a9004>}
[618] polling_check.R<R:a9009>:OUTPUT: == received outcome: 4
[1303] polling_check.R<R:a9003>:OUTPUT: == received outcome: 4
[400] polling_check.R<R:a900b>:OUTPUT: == received outcome: 4
[1082] polling_check.R<R:a9005>:OUTPUT: == received outcome: 4
[1194] polling_check.R<R:a9004>:OUTPUT: == received outcome: 4
[860] polling_check.R<R:a9007>:OUTPUT: == received outcome: 4
[1417] polling_check_live.Checker<Checker:a9002>:OUTPUT: !! L2 timeout receiving outcome by
all pollees 0 {<R:a9004>, <R:a900a>, <R:a9007>, <R:a9009>, <R:a9003>, <R:a9005>, <R:a900c>,
<R:a9006>, <R:a900b>, <R:a9008>} <R:a9008>
[511] polling_check.R<R:a900a>:OUTPUT: == received outcome: 4
[974] polling_check.R<R:a9006>:OUTPUT: == received outcome: 4
[733] polling_check.R<R:a9008>:OUTPUT: == received outcome: 4
[291] polling_check.R<R:a900c>:OUTPUT: == received outcome: 4
[1438] polling_check_live.Checker<Checker:a9002>:OUTPUT: ~~ polling ended. checking safety:
True True
```

Notice the last process, `<R:a9008>`, printed in the 3 lines reporting (L2) timeout; it shows a witness for violation of the `each` check on lines 25–27 in Fig. 3, printed at the end of line 28. When we increased the timeout for `'q-o'` to 0.01 s, no (L2) timeout was reported in all dozens of runs checked. When we added message loss rate of 10%, we saw some runs reporting total timeout, and some runs reporting even all three timeouts.

Overall, we have used the checking framework in implementation, testing, debugging, simulation, and analysis of many well-known distributed algorithms, and in developing their high-level executable specifications. This includes a variety of algorithms for distributed mutual exclusion and distributed consensus written in DistAlgo [23,25,28,30], especially including over a dozen well-known algorithms and variants for classical consensus and blockchain consensus [26]. Use of DistAlgo has helped us find improvements to both correctness and efficiency of well-known distributed algorithms, e.g., [23,25,28].

We have also used the framework in other research, e.g., [24], and in teaching distributed algorithms and distributed systems to help study and implement many more algorithms. DistAlgo has been used by hundreds of students in graduate and undergraduate courses in over 100 different course projects, implementing and checking the core of network protocols, distributed graph algorithms, distributed coordination services, distributed hash tables, distributed file systems, distributed databases, parallel processing platforms, security protocols, and more [29].

The algorithms and systems can be programmed much more easily and clearly compared to using conventional programming languages, e.g., in 20 lines instead of 200 lines, or 300 lines instead of 3000 lines or many more. Systematic methods for checking these algorithms and implementations has been a continual effort.

Additional information is available at http://distalgo.cs.stonybrook.edu/tutorial.

8 Related Work

Francalanza et al. broadly surveyed runtime verification research related to distributed systems [8]. Here, we focus on aspects related to DistAlgo.

Global Property Detection. Many algorithms have been developed to detect global properties in distributed systems, e.g., [9,15]. These algorithms vary along several dimensions. For example, many consider only the happened-before ordering [17]; others also exploit orderings from approximately-synchronized real-time clocks [38]. Some can detect arbitrary predicates; other are specialized to check a class of properties efficiently. Many use a single checker process (as in our example); others use a hierarchy of checker processes, or are decentralized, with the locus of control moving among the monitored processes. DistAlgo's high-level nature makes it very well-suited for specifying and implementing all such algorithms.

Efficient Invariant Checking. Runtime checking of invariants, in centralized or distributed systems, requires evaluating them repeatedly. This can be expensive for complex invariants, especially invariants that involve nested quantifiers. We used incrementalization for efficient repeated evaluation of predicates in the contexts of runtime invariant checking and query-based debugging for Python programs [10, 11]. We later extended our incrementalization techniques to handle quantifications in DistAlgo programs [29].

Centralization. Due to the difficulty of runtime checking of truly distributed systems, some approaches create centralized versions of them. We have developed a source-level centralization transformation for DistAlgo that produces a non-deterministic sequential program, well-suited to simulation and verification. In prior work, we developed bytecode-level transformations that transform a distributed Java program using Remote Method Invocation (RMI) into a centralized Java program using simulated RMI [39]. Minha [32] takes another approach to centralization of distributed Java programs, by virtualizing multiple Java Virtual Machine (JVM) instances in a single JVM and providing a library that simulates network communication.

DistAlgo Translators. Grall et al. developed an automatic translation from Event-B models of distributed algorithms to DistAlgo [12]. Event-B is a modeling language adapted to verification of distributed algorithms. They chose DistAlgo as the target language because "Its high-levelness makes DistAlgo closer to the mathematical notations of Event-B and improves the clarity of DistAlgo programs." We developed a translator from DistAlgo to TLA+, allowing verification tools for TLA+ to be applied to the translations [27].

Conclusion. We have presented a general, simple, and complete framework for runtime checking of distributed algorithms. The framework, as well as the algorithms and properties to be checked, are written in a high-level language that is both completely precise and directly executable. A challenging problem for future work is a powerful, commonly accepted language for higher-level, declarative configurations for checking distributed algorithms and systems.

References

1. Birman, K., Joseph, T.: Exploiting virtual synchrony in distributed systems. In: Proceedings of the 11th ACM Symposium on Operating Systems Principles, pp. 123–138. ACM Press, November 1987
2. Birman, K., Malkhi, D., Renesse, R.V.: Virtually synchronous methodology for dynamic service replication. Technical report MSR-TR-2010-151, Microsoft Research (2010)
3. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Trans. Comput. Syst. (TOCS) **5**(1), 47–76 (1987)
4. Chand, S., Liu, Y.A.: What's live? Understanding distributed consensus. Computing Research Repository arXiv:2001.04787 [cs.DC], January 2020. http://arxiv.org/abs/2001.04787

5. Chand, S., Liu, Y.A., Stoller, S.D.: Formal verification of multi-Paxos for distributed consensus. In: Fitzgerald, J., Heitmeyer, C., Gnesi, S., Philippou, A. (eds.) FM 2016. LNCS, vol. 9995, pp. 119–136. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-48989-6_8
6. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985)
7. Fokkink, W.: Distributed Algorithms: An Intuitive Approach. MIT Press, Cambridge (2013)
8. Francalanza, A., Pérez, J.A., Sánchez, C.: Runtime verification for decentralised and distributed systems. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 176–210. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_6
9. Garg, V.K.: Elements of Distributed Computing. Wiley, New York (2002)
10. Gorbovitski, M., Rothamel, T., Liu, Y.A., Stoller, S.D.: Efficient runtime invariant checking: a framework and case study. In: Proceedings of the 6th International Workshop on Dynamic Analysis, pp. 43–49. ACM Press (2008)
11. Gorbovitski, M., Tekle, K.T., Rothamel, T., Stoller, S.D., Liu, Y.A.: Analysis and transformations for efficient query-based debugging. In: Proceedings of the 8th IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 174–183. IEEE CS Press (2008)
12. Grall, A.: Automatic generation of DistAlgo programs from Event-B models. In: Raschke, A., Méry, D., Houdek, F. (eds.) ABZ 2020. LNCS, vol. 12071, pp. 414–417. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-48077-6_34
13. Hawblitzel, C., et al.: IronFleet: proving practical distributed systems correct. In: Proceedings of the 25th Symposium on Operating Systems Principles, pp. 1–17. ACM Press (2015)
14. Kane, C., Lin, B., Chand, S., Stoller, S.D., Liu, Y.A.: High-level cryptographic abstractions. In: Proceedings of the ACM SIGSAC 14th Workshop on Programming Languages and Analysis for Security. ACM Press, London, November 2019
15. Kshemkalyani, A., Singhal, M.: Distributed Computing: Principles, Algorithms, and Systems. Cambridge University Press, Cambridge (2008)
16. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Softw. Eng. **3**(2), 125–143 (1977)
17. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
18. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
19. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
20. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
21. Lin, B., Liu, Y.A.: DistAlgo: a language for distributed algorithms (2014). http://github.com/DistAlgo. Accessed March 2020
22. Liskov, B., Cowling, J.: Viewstamped replication revisited. Technical report MIT-CSAIL-TR-2012-021, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge (2012)
23. Liu, Y.A.: Logical clocks are not fair: what is fair? A case study of high-level language and optimization. In: Proceedings of the Workshop on Advanced Tools, Programming Languages, and Platforms for Implementing and Evaluating Algorithms for Distributed Systems, pp. 21–27. ACM Press (2018)

24. Liu, Y.A., Brandvein, J., Stoller, S.D., Lin, B.: Demand-driven incremental object queries. In: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming, pp. 228–241. ACM Press (2016)
25. Liu, Y.A., Chand, S., Stoller, S.D.: Moderately complex Paxos made simple: high-level executable specification of distributed algorithm. In: Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming, pp. 15:1–15:15. ACM Press, October 2019
26. Liu, Y.A., Stoller, S.D.: From classical to blockchain consensus: what are the exact algorithms? In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, July–August 2019, pp. 544–545. ACM Press (2019)
27. Liu, Y.A., Stoller, S.D., Chand, S., Weng, X.: Invariants in distributed algorithms. In: Proceedings of the TLA+ Community Meeting, Oxford, U.K. (2018). http://www.cs.stonybrook.edu/~liu/papers/DistInv-TLA18.pdf
28. Liu, Y.A., Stoller, S.D., Lin, B.: High-level executable specifications of distributed algorithms. In: Richa, A.W., Scheideler, C. (eds.) SSS 2012. LNCS, vol. 7596, pp. 95–110. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33536-5_11
29. Liu, Y.A., Stoller, S.D., Lin, B.: From clarity to efficiency for distributed algorithms. ACM Trans. Program. Lang. Syst. **39**(3), 12:1–12:41 (2017)
30. Liu, Y.A., Stoller, S.D., Lin, B., Gorbovitski, M.: From clarity to efficiency for distributed algorithms. In: Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 395–410. ACM Press (2012)
31. Lynch, N.A.: Distributed Algorithms. Morgan Kaufman, San Francisco (1996)
32. Machado, N., Maia, F., Neves, F., Coelho, F., Pereira, J.: Minha: large-scale distributed systems testing made practical. In: Felber, P., Friedman, R., Gilbert, S., Miller, A. (eds.) 23rd International Conference on Principles of Distributed Systems (OPODIS 2019). LIPIcs, vol. 153, pp. 11:1–11:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019)
33. McMillan, K.L., Padon, O.: Ivy: a multi-modal verification tool for distributed algorithms. In: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 190–202. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_12
34. Microsoft Research: the TLA toolbox. http://lamport.azurewebsites.net/tla/toolbox.html. Accessed 27 Apr 2020
35. Oki, B.M., Liskov, B.H.: Viewstamped replication: a new primary copy method to support highly-available distributed systems. In: Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing, pp. 8–17. ACM Press (1988)
36. Padon, O., Losa, G., Sagiv, M., Shoham, S.: Paxos made EPR: decidable reasoning about distributed protocols. Proc. ACM Program. Lang. **1**(OOPSLA), 108:1–108:31 (2017). Article no. 108
37. Raynal, M.: Algorithms for Mutual Exclusion. MIT Press, Cambridge (1986)
38. Stoller, S.D.: Detecting global predicates in distributed systems with clocks. Distrib. Comput. **13**(2), 85–98 (2000). https://doi.org/10.1007/s004460050069
39. Stoller, S.D., Liu, Y.A.: Transformations for model checking distributed java programs. In: Dwyer, M. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 192–199. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45139-0_12