Ilmenau University of Technology
Department of Computer Science and Automation
Institute for Applied Computer Science and Media Communications
Distributed Systems and Operating Systems Group

Master Thesis

A specification language for distributed algorithms

Submitted by:
Felix Wiemuth

Advisors:	Prof. Dr.-Ing. habil. Winfried Kühnhauser Dr.-Ing. Peter Amthor
Program:	Computer Science
Matriculation Number:	48979
Submission Date:	11th December 2018
Successful final exam date:	30th January 2019

URN:	urn:nbn:de:gbv:ilm1-2019200152
DOI:	10.22032/dbt.38242

Abstract

Non-functional requirements such as termination or fault tolerance play an important role in the design of distributed algorithms. Static analyses allow the consideration of such properties early on in the design process. To what extent certain analysis goals can be reached largely depends on the specification language in use. The goal of this work is to describe a specification language for distributed algorithms which allows for static analyses of non-functional properties, where the focus is on termination.

Zusammenfassung

Nichtfunktionale Eigenschaften wie Terminierung oder Fehlertoleranz spielen eine bedeutende Rolle beim Design verteilter Algorithmen. Statische Analysen erlauben die Betrachtung solcher Eigenschaften schon während des Entwicklungsprozesses. Inwieweit bestimmte Analyseziele erreicht werden können, hängt bedeutend von der verwendeten Spezifikationssprache ab. Das Ziel dieser Arbeit ist die Beschreibung einer Spezifikationssprache für verteilte Algorithmen, welche sich für statische Analysen auf nichtfunktionale Eigenschaften eignet. Der Fokus liegt dabei auf Terminierung.

Contents

1 Introduction **1**

2 Preliminaries **3**

2.1 Distributed algorithms . . . 3
2.2 Timing models . . . 4
2.3 Failures . . . 5

3 Language design **7**

3.1 Event-driven formulation of distributed algorithms . . . 7
3.2 The input-action language . . . 8
3.2.1 Spontaneous actions . . . 9
3.2.2 Suitability of the input-action language . . . 9
3.2.3 A semantic model for the input-action language . . . 14
3.2.4 A syntax for the input-action language . . . 16
3.3 Termination analysis . . . 17
3.3.1 A termination criterion . . . 17
3.3.2 The message flow graph . . . 18
3.3.3 Correctness of the termination criterion . . . 21
3.3.4 Spontaneous actions revisited . . . 24
3.3.4.1 Simulating spontaneous actions . . . 24
3.3.4.2 External messages as an alternative . . . 24
3.3.5 Applying the termination criterion . . . 25
3.3.6 Approximating the message flow graph . . . 25
3.3.6.1 The message type graph . . . 26
3.3.6.2 Precision of the message type graph . . . 28
3.3.6.3 Better approximations . . . 29
3.3.6.4 Practicability of the message type graph . . . 29
3.3.7 Improving precision . . . 30
3.3.7.1 Limitations of the message flow graph . . . 30
3.3.7.2 Harmless cycles . . . 31
3.3.7.3 Limited edges . . . 32
3.3.7.4 Inferring termination for a ring algorithm . . . 33
3.4 Conclusion . . . 35

4 Expressivity **39**
4.1 Timers . . . 39
4.1.1 Syntax . . . 39
4.1.2 Semantics . . . 40
4.1.2.1 Time in the asynchronous model . . . 40
4.1.2.2 Simulation . . . 41
4.2 Evaluation . . . 42
4.2.1 A fault tolerant two-phase commit protocol . . . 42
4.2.2 Termination analysis . . . 44
4.2.3 Spontaneous actions . . . 44
4.2.4 Real systems . . . 46
4.3 Conclusion . . . 46

5 Usability **49**
5.1 Deficiencies of the input-action language . . . 49
5.2 Protocol stages . . . 50
5.2.1 Syntax . . . 50
5.2.2 Semantics . . . 51
5.3 Evaluation . . . 53
5.3.1 The two-phase commit protocol with stages . . . 53
5.3.2 Structuring tasks using stages . . . 56
5.3.3 Replacing timers with stage timeouts . . . 57
5.3.4 Managing unexpected messages . . . 57
5.3.5 Stages and analyzability . . . 58
5.4 Conclusion . . . 59

6 Implementation **61**
6.1 Towards a programming language . . . 61
6.2 Syntax . . . 61
6.2.1 Message type definitions . . . 62
6.2.2 Task definition . . . 62
6.3 Compiler . . . 64
6.3.1 Architecture . . . 64
6.3.2 Lexing and parsing . . . 64
6.3.3 Runtime system . . . 64
6.3.4 Code generation . . . 65
6.3.5 Error reporting . . . 66
6.4 Simulation . . . 66
6.5 Termination analysis . . . 67
6.6 Tests . . . 67
6.7 User interface . . . 68
6.8 Example . . . 68
6.8.1 The two-phase commit protocol implemented in JIAL . . . 68
6.8.2 Compilation to Java . . . 69
6.8.3 Simulation . . . 73
6.9 Practical application . . . 73
6.10 Conclusion . . . 73

7 Conclusion **75**

Bibliography **77**

CHAPTER 1

Introduction

Distributed systems play an important role in many aspects of today's life. They provide reliable cloud storage, safe transportation, real-time communication and secure transactions. Whenever there is a computation involving multiple physically separated components, we are talking about a distributed system. A distributed algorithm is the software part of this system.

Reliability, safety, real-time behaviour and security are just some out of many non-functional properties important for distributed systems. Their importance endorses an early consideration in the design process of a distributed algorithm. Here static analyses are an important tool, automatically determining interesting properties. Which properties can be automatically inferred greatly depends on the specification language and its paradigms used to formulate algorithms. In general, it is more difficult to analyze an algorithm formulated in a very low-level language than an algorithm formulated in a language with high-level abstractions accounting for the specifics of distributed systems.

An important non-functional property is termination. Imagine for example a redundant system in an airplane which has to make decisions based on the opinions of multiple sensors and computers. Algorithms for this purpose can be more complex than just implementing a majority decision and it would be fatal if the decision process does not terminate.

While termination is similarly important to other types of computations, the situation for distributed algorithms is special and complex. First of all, what does it mean for a distributed algorithm to terminate? As the global interaction between the algorithm's components has to be taken into account, the situation is fundamentally different from just local computations.

Contribution and outline

The goal of this work is to describe a specification language for asynchronous distributed algorithms which allows for static termination analysis. At the same time, the language should be easy to use and have prospects of being suitable for the analyses on further non-functional properties.

As a starting point, we take an event-driven language which was introduced by Barbosa [Bar96] as a high-level language for the formulation of distributed algorithms. It is based on the guarded command language by Dijkstra [Dij75], allowing for the expression of non-determinism which is an important factor in asynchronous distributed algorithms.

In the main part of the work (Chapter 3), we show that the language is indeed suitable for termination analysis. To do so, we first define a precise semantic model for the language within an asynchronous setting. We then develop a termination criterion and prove its correctness. Termination is defined as the absence of any further activity, where we factor out local computations to be able to focus on global interaction.

To tackle the secondary goals, suitability for the analyses on further non-functional properties and usability, in chapters 4 and 5, we present two extensions to the language. The first integrates timers as a way of expressing the detection of failures. The second accounts for a common design pattern of distributed algorithms by allowing to structure protocols into *stages*, facilitating formulation also with regard to timeouts. We show that both extensions are compatible with the termination analysis. In fact, they can be expressed in the same semantic model.

As a practical evaluation of our work we integrate a general-purpose programming language into the before abstract specification language, resulting in a full programming language for which we implement a compiler (Chapter 6). Algorithms can then be simulated on a single machine as well as executed in a real distributed setting. In addition, we implement a tool which checks the specification of an algorithm for termination based on the termination criterion we develop.

CHAPTER 2

Preliminaries

In this chapter, we give an overview on the background on distributed systems and algorithms required to follow this work. In the process we describe the model of distributed algorithms our considerations are based on.

In the literature, one can find different definitions of distributed systems and distributed algorithms (see for example [Bar96], [Lyn96], [TV07]). Very generally, one can regard a *distributed system* as a collection of processors connected by a network, interacting to achieve a common goal. The software running on the processors of such a system is a *distributed program*, or, when focusing on the concepts rather than technical aspects, a *distributed algorithm*. Definitions differ for example in the interpretation of the term *distributed*. Does the system have to be composed of physically distributed machines, or can also multiple communicating processes on a single machine be regarded as a distributed algorithm? It turns out that algorithms in different distributed settings have many properties in common but can also differ fundamentally. For example, in a faultless setting, for an algorithm communicating via messages it conceptually makes no difference whether the components run on the same machine or are located far away from each other. Regarding failures, however, there is a difference: on a local machine, failures are not independent (for example, if the machine crashes, all tasks fail) while one can assume tasks on different machines to fail independently. Further, if a network is involved, there are types of failures that usually do not exist locally (like the loss of a message).

As our studies are of a more general nature, there is no need to specifically restrict these terms. Instead, we use a very abstract definition of distributed algorithms.

2.1 Distributed algorithms

We abstractly define a *distributed algorithm* to be a set of *tasks*, which represent the local components of the algorithm. A task can perform local computations and keep a local state. Furthermore, tasks can communicate with other tasks using *messages*, sent as part of their computation. We leave open the form of addressing used when sending messages and what information about the origin of a message can be obtained by the receiver. Different ways of specifying receivers of messages exist, including direct addressing (by ID), broadcast, multicast or even anonymous replies. In the literature (e.g. [Bar96]), addressing is sometimes realized by sending messages through *channels* which connect tasks. However,

this makes unnecessary assumptions about the network topology and also restricts the possible level of anonymity. Abstractly, this kind of addressing can be seen as using pseudonyms, where the channel names are pseudonyms for tasks. Using pseudonyms does not allow for full anonymity though: tasks can learn about each other by observing the relation between sending and receiving messages on different channels. Not limiting addressing this way allows for more anonymous forms, for example replying to broadcast messages without disclosing one's own ID.

The distribution of messages between tasks according to the different forms of addressing is assumed to be handled by the underlying network. Except for a few basic assumptions specified next, we completely abstract from the nature of this network, as it is of no relevance to our considerations.

2.2 Timing models

The way messages are delivered and processed depends on the timing model of the underlying system, including the network. Here we distinguish between two basic models, the *synchronous* and the *asynchronous* model. The properties of these models have a significant influence on how distributed algorithms can be formulated.

In the synchronous model, there is a global clock synchronizing communication and processing. The algorithm runs in rounds of fixed time intervals. Each round consists of two steps: delivering of messages and processing. At the beginning of each time interval, all messages sent during the computation in the previous round are delivered to their destination. Then, each task processes all the messages received and possibly sends new messages. This execution in lock step implies that delivery of messages as well as processing are guaranteed to complete in known, finite time intervals. The synchronous model therefore makes strong assumptions which are, however, indeed required in many practical applications.

The asynchronous model on the other hand makes only very weak assumptions. Here tasks execute their programs independent of each other and messages can be sent at any time. There are no assumptions about when and in which order messages are delivered. The only guarantee is that messages are delivered after a finite amount of time. Each task has a buffer for incoming messages with infinite capacity which it can inspect and modify at any time. Note that in contrast to the synchronous model, the speed of computation at tasks is irrelevant as the model has no notion of time and communication can take arbitrarily long.

We call an algorithm formulated in the synchronous model a *synchronous algorithm* and an algorithm formulated in the asynchronous model an *asynchronous algorithm*. Of course, also algorithmic models between the two extremes of full synchronicity and full asynchronicity can be considered. In this work, however, we only study asynchronous algorithms.

We note that the non-deterministic communication behaviour of the asynchronous model can, but does not have to, lead to non-deterministic algorithms whereas algorithms in the synchronous model with deterministic local computations are always deterministic (within the faultless model). Another important

characteristic of the synchronous model is that the absence of messages conveys information whereas in the asynchronous model, due to the unpredictable delivery time, the absence of a message can never be detected.

With its strong assumptions, especially the deterministic communication behaviour, the synchronous model is certainly the one easier to program in while the asynchronous model is more natural regarding its implementation in practical systems. However, it can be shown that both models are equivalent, i.e., for each algorithm correct in the synchronous model (regarding a certain specification) there exists an algorithm correct in the asynchronous model and vice versa [Bar96, Chapter 3.3].

2.3 Failures

While the above models describe an ideal situation, different kinds of failures can be encountered in the execution of distributed algorithms. Generally, there are two types of failures: message failures and task failures. A message failure can consist of a message not being delivered, a message being delivered more than once, or, if also byzantine failures are considered, altering of messages or the creation of messages that have not been sent by tasks. Task failures come in three levels of complexity. In the simplest form, called *fail-stop*, a task just stops completely, i.e., does not send any messages for the rest of the algorithm's execution. A *fail-stop-return* failure in contrast describes the intermittent unavailability of a task. This is significantly different from fail-stop as the restart of a task has to be considered. Finally, a *byzantine* failure describes the situation where a task does not behave according to its program, i.e., it sends arbitrary messages.

Note that from the view of other tasks, a task failure can only be observed via the messages it sends or does not send. Consequently, it cannot be distinguished whether the cause of a lost or changed message was a task or message failure. In the synchronous model, a missing message (whatever the reason) is detected in the processing step of a round. In the asynchronous model, the loss of a message or the failure of a task can never be detected with full certainty because of the unpredictable message delivery time.

CHAPTER 3

Language design

In this chapter, we introduce a semantic model for an event-driven language for asynchronous distributed algorithms, based on the guarded command language by Dijkstra [Dij75] and its adaptation to distributed algorithms by Barbosa [Bar96]. We show that languages based on this model lend themselves to static termination analyses.

3.1 Event-driven formulation of distributed algorithms

A commonly used way of implementing network communication in programming languages is through send and receive primitives. This paradigm has the disadvantage of receive statements being entangled with local program logic, making it difficult to reason about such programs.

This problem can be avoided by using an event-driven paradigm where control flow is based on certain events triggering certain actions. In the context of distributed algorithms, the most important event is the reception of a message. As an example of an event-driven formulation of a distributed algorithm consider a client-server architecture where the server offers certain services to multiple clients. Each service can be accessed by sending a special message. Now the server's program can be formulated as a set of actions, each to be executed on the reception of the corresponding message.

The guarded command language

An event-driven language, based on actions being performed on state changes (which can be interpreted as events), is the *guarded command language* introduced by Dijkstra [Dij75]. It was not specifically designed for distributed or parallel programs but its paradigm turns out to be very useful in this domain. The essence of the language is the guarded command construct which couples a command with a condition (guard) for its execution. More precisely, a guard is a Boolean expression over the current state of the program (e.g. its variables) and a command is a sequence of statements transforming this state. Guarded commands are grouped into alternative and repetitive constructs which form statements of an arbitrary imperative language. We will only make use of the repetitive construct and programs will consist of only one such statement. A program can then be regarded as a set of guarded commands. It is executed

by repeatedly choosing and executing a command with a satisfied guard. Note that this allows for the expression of non-determinism: if multiple commands are eligible for execution, we have not specified which should be executed.

The language was originally introduced for *predicate transformer semantics*, which is based on Hoare logic, to reason about programs. This already suggests that it might be easier to derive certain properties when algorithms are formulated in this more abstract and quite simple paradigm.

We will see that the guarded command paradigm is useful for the formulation of distributed algorithms in general and eventually allows for a termination analysis regarding the "global behaviour" of an algorithm.

3.2 The input-action language

We now describe a specification language for asynchronous distributed algorithms based on the guarded command paradigm. With regard to its event-driven behaviour with messages (*inputs*) triggering *actions*, we call it the *input-action language*. It was introduced in a similar form by Barbosa in [Bar96].

The main component of the language, forming a self-contained module, is the *task*, describing a local program with a local state and an *input buffer* for messages received from other tasks. A collection of tasks describes a distributed algorithm. We do not have to further specify what the state of a task is, in a simple form it can consist of the values of the local variables. A task's logic is described by a set of *input-action pairs*. Similar to a guarded command, an input-action pair couples a condition with an action. However, instead of just being a condition on local state, here a guard in addition always contains a matcher on incoming messages (hence the term *input*). A matcher is simply a Boolean condition on messages' content as well as their metadata (e.g. their origin). An action is a sequence of statements transforming the local state of a task and possibly producing messages to be sent to other tasks. Except for primitives to send messages, the concrete nature of statements is irrelevant to our considerations. The goal is to have a *specification language* which allows for static analyses even without fully implementing an algorithm.

The semantics of the language follows the asynchronous timing model described in Section 2.2 and can be intuitively described as follows. Whenever, in any task, the guard of an input-action pair is fulfilled, that is, a matching message is present in the input buffer and the potential condition on local state is satisfied, the corresponding action is executed with such a matching message, which is then removed from the buffer. Messages sent during the execution of an action are added to the destinations' input buffers. Actions are executed atomically, that is, only after the execution of an action is completed, guards are checked again for further actions to be executed. When multiple actions are eligible for execution, one is chosen non-deterministically. Furthermore, there is no given order in which the tasks of an algorithm perform actions.

A language similarly based on the guarded command paradigm is Promela, a verification modeling language which can be used to formulate distributed algorithms. Promela programs can be verified with the Spin model checker

using standard Linear Temporal Logic (LTL) [Hol97]. While this approach is quite general and can be used to verify a variety of non-functional properties (including for example the absence of deadlocks), its approach is different from ours in that it is based on *dynamic* analysis (which requires the model to have certain bounds to be decidable) whereas we want to develop a *static* analysis, resulting in a purely syntactic termination criterion.

3.2.1 Spontaneous actions

Note that by requiring a guard to match a message and thus not allowing it to solely depend on local state, we deviate from the original guarded command paradigm. Here we understand a task as "a reactive (or message-driven) entity, in the sense that normally it only performs computation (including the sending of messages to other tasks) as a response to the receipt of a message from another task" [Bar96, p. 14]. This is how Barbosa introduces tasks in the context of *message-passing programs* which form the basis for the model of asynchronous distributed algorithms the input-action language is based on. An exception to this rule is made to initialize an algorithm: for this purpose, a task may once spontaneously send messages. We handle initialization differently, simply by assuming initial messages in the input buffers or via *external messages*. This is why, with respect to initialization, there is no need for spontaneous messages or actions.

Nevertheless, some algorithms in [Bar96] actually make use of guards on local state without receiving a message. This is problematic with regard to the termination analysis we want to establish. If tasks can initiate actions just depending on local state, a major part of a termination analysis would consist of analyzing whether tasks on their own terminate from their current state. We want to exclude this aspect from our termination analysis and focus on the interaction between tasks. For that reason, we use the original reactive model, where each action has to consume a message.

The necessity and consequences of this restriction are further discussed in sections 3.3.2 and 3.3.4. However, it shall already be noted that by shifting activation to *external messages*, spontaneous actions can be modelled without affecting the termination analysis.

3.2.2 Suitability of the input-action language

The goal of this section is to get familiar with the usage of the input-action language and to get an intuition why it is suitable for the formulation of distributed algorithms.

Incorporating the guarded command paradigm into a specification language for distributed algorithms results in several advantages over a language based on send and receive primitives. First, a task is no more a single sequential program with loops and branches, listening for messages at certain points. Instead, control flow is driven by the messages received from other tasks. This allows for a more natural design process of distributed algorithms: one does not have to think

about in which part of the program a certain message should be received but rather what effect the reception of a certain message should have. Second, the implicit buffering of messages until input-action pairs are ready to process them allows to abstract from the order in which messages arrive. This acknowledges the fact that no such order is given by the asynchronous timing model anyway.

Algorithm 3.1 (further explained below) gives an impression on how the input-action language is used to formulate distributed algorithms. We use a pseudo code syntax which should intuitively fit the language described so far. It will be formalized in Section 3.2.4.

The headline of each task consists of a name and some identifiers for the instances of that task to be part of the algorithm. Also sets of identifiers to be used for addressing can be specified. In this case, the algorithm consists of one task "Coordinator" (C) and n tasks "Participant" ($\mathrm{P}_1, \ldots, \mathrm{P}_n$). The set P denotes the set of all participants.

Each task is described by some local variable declarations, followed by a number of input-action pairs. An input-action pair is introduced by the `input` keyword, followed by a guard which consists of a *message matcher* and possibly additional conditions (specified in a *when clause* as explained later). The corresponding action is represented by a code block enclosed in curly braces.

Messages consist of a name and a list of parameters. For example, the message `vote(abort)` has the name "vote" and a parameter "abort". In an input-action pair's action, parameters are bound to the variables specified in the guard's message matcher (for example, a guard with message matcher `vote(x)` can match a message `vote(abort)` in which case `x` is bound to "abort").

Messages are sent in actions using `send` and `reply` statements. A `reply` is always addressed to the origin of the message currently being processed. The `send` statement can specify a single task, like C, or a set of tasks, like P, as destination. Note that in our examples we use direct addressing by ID, though in general, the `send` statement does not have to be restricted to this form.

Algorithm 3.1 implements the *two-phase commit protocol* (originally described in [Gra78]) the goal of which is to decide whether a distributed transaction should be committed or aborted. To this end, in a *voting phase* a designated *coordinator* asks each *participant* whether it would be able to commit its local part of the transaction. Only after all participants (and the coordinator itself) have voted to commit, the coordinator sends the OK to commit, otherwise it advises the participants to abort (this is the *commit phase*). Note that for each "abort" vote the coordinator receives, it sends out `abort()` messages to all participants and each task will execute `t.abort()`. We accept this here and assume the operation to be idempotent. The algorithm is initialized by sending an `init()` message to the coordinator task.

To demonstrate how additional conditions can be added to a guard, Algorithm 3.2 reformulates the Coordinator task of Algorithm 3.1. The `when` clause allows to specify any conditions on the message and local state for the action to be eligible for execution. As can be seen in this example, the complexity of single actions can be reduced by splitting them into multiple input-action pairs, moving conditionals to guards.

Coordinator: { C }

```
Variables
    t // local part of transaction
    v // own vote (commit/abort)
    count := 0 // commit count

input init() {
    if v = abort then
        send abort() to P
        t.abort()
    else
        send vote_request() to P
    end
}

input vote(x) {
    if x = abort then
        send abort() to P
        t.abort()
    else
        count++
        if count = n then
            send commit() to P
            t.commit()
        end
    end
}
```

Participant: P = { $P_1, \ldots, P_n$ }

```
Variables
    t // local part of transaction
    v // own vote (commit/abort)

input vote_request() {
    reply vote(v)
}

input abort() {
    t.abort()
}

input commit() {
    t.commit()
}
```

Algorithm 3.1: The two-phase commit protocol

Coordinator: { C }

```
Variables
    t // local part of transaction
    v // own vote (commit/abort)
    count := 0 // commit count

input init() {
    if v = abort then
        send abort() to P
        t.abort()
    else
        send vote_request() to P
    end
}

input vote(abort) {
    send abort() to P
    t.abort()
}

input vote(commit) when count < n − 1 {
    count++
}

input vote(commit) when count = n − 1 {
    send commit() to P
    t.commit()
}
```

Algorithm 3.2: The two-phase commit protocol (alternate formulation of Coordinator task of Algorithm 3.1 using the `when` clause)

However, the real power of guards on local state lies in holding back messages until a suitable moment of processing. As an example, consider a server building up a list, the elements of which are provided by different clients. The list is to be constructed in order (imagine a linked list) and elements should be added as soon as possible. This can be useful if the elements have to be processed in order and a greedy processing is desired. With a guard on the current construction state of the list, this can be implemented easily: the position i is added to the list as soon as the previous position has been added and the element itself is available (a client has sent it). Algorithm 3.3 implements the described functionality. Note that here message parameters are in addition given a type. The first input-action pair constructs the list by receiving elements. However, the processing of an element is delayed until the list has been constructed up to the position just before that element. That is, only after the list has reached the right length, the message is processed and the new element is added. The second input-action pair allows clients to query elements. Requests are held back until the requested element is available. In a similar fashion, the third input-action pair returns the sum of elements 0 through i. Here it is necessary that all these elements have been received yet and by construction, a necessary condition for this is the list having length greater than i. Now clients can request a sum and the server automagically responds when the required elements are available.

The dynamic construction of a list is an example of how processing of messages can be delayed to abstract from the actual order in which messages are received. By making manual collection and organization of messages superfluous, this allows for much shorter and easier-to-read code.

List

```
Variables
    list := List.empty() // positions: 0, 1, 2, ...

input list_element(int pos, String s) when list.length() = pos {
    list.add(s)
}

input get_element(int i) when list.length() > i {
    reply element(list.get(i))
}

input get_sum(int i) when list.length() > i {
    reply sum(list.get(0) + ··· + list.get(i))
}
```

Algorithm 3.3: Making use of the `when` clause: ordered construction of a list

The remainder of this chapter is devoted to showing that the input-action language is useful when wanting to analyze an algorithm for termination. We do this by developing a termination analysis and proving its correctness. For this purpose, we need a clear understanding of the semantic model which is the basis for the (abstract) input-action language we have so far described only intuitively.

Therefore, we next introduce a formal model which also draws the lines of possible concrete languages and their semantics.

3.2.3 A semantic model for the input-action language

The goal of the following formalization is a model that precisely describes the global behaviour of asynchronous distributed algorithms in the sense of the input-action paradigm while abstracting from details of tasks' local activity. The idea is to represent (the configuration of) a task by its local state and a set of received, but not yet processed messages (the input buffer). A task's logic is described by a set of actions and a function selecting the currently applicable actions. An action is represented by a state transition function which consumes a message, produces new messages and a new task state. The selection function represents the guards of all input-action pairs by matching messages from the input buffer with applicable actions. An execution of an algorithm then consists of repeatedly letting tasks take a step, that is, apply an action on a matching message according to the selection function, updating its state and adding the resulting messages to other tasks' input buffers. The order in which tasks take steps is arbitrary which represents the lack of order in the asynchronous timing model.

Note that though messages are added to the destinations' input buffers directly, the asynchronous timing model's property of arbitrarily delaying and changing the order in which messages are delivered is appropriately represented. As tasks cannot directly inspect messages before also processing them and the choice of which task takes a step next is arbitrary, the possible executions would be the same if messages were temporarily kept by a "delivery component" before being added to the input buffers.

In the following, we assume a fixed number n of tasks and let $I = \{ 1, \ldots, n \}$ denote the set of *task identifiers*. Further, let Q denote a (possibly infinite) set of *task states* (the union of all tasks' possible states). We model all possible messages by a (possibly infinite) set $\mathcal{M}$.

We start the formalization by defining actions to be transition functions on the set of states Q, with a message as input and n sets of messages as output, representing the messages to be sent to each of the n tasks. Note that for technical reasons (an extension in Chapter 4) the model allows any finite amount of messages to be sent per step, though usually one can restrict this to one message per destination. It does not make any difference to the properties we study, as long as a bound on the overall number of messages sent in a step is given.

Definition 3.1. An *action* is a function $\delta \colon Q \times \mathcal{M} \to Q \times \mathcal{P}(\mathcal{M})^n$. The set of all actions is denoted by A. ◁

A task can then be defined as a function selecting pairs of actions and messages given the current state and input buffer. The resulting pairs represent all non-deterministic possibilities of a task to take a step, that is, to perform an action on a message from the input buffer. In accordance with the idea of the input-action

paradigm (a guard only has a message *matcher* and does not inspect the input buffer directly), the selection of a message must not depend on other messages in the input buffer. Further, the set of all possible actions the task can ever select must be finite (which is of course given when modelling a task from a finite set of input-action pairs). The function can be understood as the *guard function* g.

Definition 3.2. A *task* is a function $g\colon Q \times \mathcal{P}(\mathcal{M}) \to \mathcal{P}(A \times \mathcal{M})$ where for all $q \in Q, M \subseteq \mathcal{M}, \delta \in A$ and $m \in \mathcal{M}$:

- $(\delta, m) \in g(q, M)$ implies $m \in M$ and $(\delta, m) \in g(q, \{m\})$.
- $(\delta, m) \in g(q, \{m\})$ implies $(\delta, m) \in g(q, \{m\} \cup M')$ for all $M' \subseteq \mathcal{M}$.
- The set $\{\, \delta \in A \mid \exists q \in Q, m \in \mathcal{M}\colon (\delta, m) \in g(q, \{m\}) \,\}$ is finite. ◁

A distributed algorithm is then simply a collection of tasks.

Definition 3.3. A *distributed algorithm* is a tuple $\mathcal{A} = (g_1, \ldots, g_n)$ of tasks g_i for $i \in I$, also written as $(g_i)_{i \in I}$. ◁

Our formalism has some parallels to *I/O automata*, which are used to formally study asynchronous distributed algorithms in [Lyn96]. An I/O automaton, representing a task, is a state machine where the transitions are associated with inputs and outputs, modelling communication between tasks. This concept can be used to show various properties of asynchronous algorithms, including safety and liveness properties. However, for our specific purposes, the described, slightly simpler model is sufficient.

To describe the dynamic behaviour of an algorithm, we first have to introduce the notion of an algorithm's *configuration*, the collectivity of the tasks' configurations. A *task configuration* consists of a task's current state $q \in Q$ and its input buffer $M \subseteq \mathcal{M}$.

Definition 3.4. A *task configuration* is a tuple $(q, M) \in Q \times \mathcal{P}(\mathcal{M})$. A *configuration* of a distributed algorithm is an n-tuple $c = (c_1, \ldots, c_n)$ of task configurations c_i for $i \in I$. The set of all algorithm configurations is denoted by C. ◁

Finally, we can describe an algorithm's execution as a sequence of successor configurations. A configuration's possible successor configurations are determined by the steps the tasks can currently take. A task can take a step if in its current state q there is an input-action pair which can process a message m from the input buffer M (that is, the task's matching function $g(q, M)$ yields at least one pair (δ, m)). The action δ is then applied to q and m resulting in a new state q' and sets $M_1^+, \ldots, M_n^+$ of new messages to be sent to each task. The algorithm's new configuration then results from the old configuration by replacing the state of the task which took a step by its new state, removing the message consumed by that task from its input buffer and for each task $i \in I$ adding the messages M_i^+ to its input buffer.

Definition 3.5. The *successor relation* $\to_{\mathcal{A}} \subseteq C \times C$ for $\mathcal{A}$ is defined by

$$\begin{aligned} &((q_1, M_1), \ldots, (q_k, M_k), \ldots, (q_n, M_n)) \\ \to_{\mathcal{A}}\ &((q_1, M_1'), \ldots, (q_k', M_k'), \ldots, (q_n, M_n')) \end{aligned}$$

$$\iff \exists k \in I, (\delta, m) \in g_k(q_k, M_k) \colon \delta(q_k, m) = ({q_k}', M_1^+, \ldots, M_n^+)$$

$$\text{where } M_i' = \begin{cases} (M_i \setminus \{\, m \,\}) \cup M_i^+ & \text{if } i = k \\ M_i \cup M_i^+ & \text{if } i \neq k\ . \end{cases}$$

The reflexive and transitive closure of $\to_{\mathcal{A}}$ is denoted by $\to_{\mathcal{A}}^*$. For configurations $c, c' \in C$ with $c \to_{\mathcal{A}} c'$, c' is called a *successor configuration* of c and $\mathcal{A}$ is said to *take a step* from c to c'. A configuration $c^* \in C$ with $c \to_{\mathcal{A}}^* c^*$ is called a *future configuration* of c. A sequence of successor configurations is also called an *execution* of $\mathcal{A}$. ◁

3.2.4 A syntax for the input-action language

In the form of an EBNF grammar, we now formalize the syntax informally introduced before and used throughout all examples. It should intuitively fit the semantic model defined in the previous section. We leave open the concrete form of identifiers, variable declarations, Boolean expressions, messages, destinations and additional statements. We also do not specify yet the concrete form of message matchers. Any additional statements (such as variable assignments, comparisons, conditionals and loops) can be added to obtain a full programming language from the more abstract specification language. For our purposes (the development of a termination analysis based on global behaviour), the given elements are sufficient.

⟨*task*⟩ ::= '`task`' ⟨*task-identifier*⟩ '`{`'
⟨*variable-declarations*⟩
{ ⟨*input-action-pair*⟩ }
'`}`'

⟨*input-action-pair*⟩ ::= ⟨*input*⟩ ⟨*action*⟩

⟨*input*⟩ ::= '`input`' ⟨*msg-matcher*⟩ ['`when`' ⟨*boolean-exp*⟩]

⟨*action*⟩ ::= '`{`' { ⟨*statement*⟩ } '`}`'

⟨*statement*⟩ ::= '`send`' ⟨*msg*⟩ '`to`' ⟨*dest*⟩
| '`reply`' ⟨*msg*⟩
| ...

In the examples, the "task header" is represented by the headline.

It is straightforward to translate a distributed algorithm formulated in this syntax into our semantic model.

3.3 Termination analysis

We now show that the input-action language lends itself to termination analysis. To this end, we first develop a termination criterion for algorithms given in the semantic model and then transfer it into an efficiently decidable syntactic criterion.

We do not consider complete termination analyses which are a field of study on their own. Instead, we investigate an aspect of termination specific to distributed algorithms: control flow loops through communication. As messages invoke actions and actions can produce new messages, an algorithm can diverge in message loops. Our goal is to statically determine whether this can be the case. To restrict ourselves to this global aspect, we make the assumption that locally, each action terminates on invocation. To obtain a full termination analysis, the termination of local actions can be checked by conventional termination analyses for sequential programs (which of course is not possible in all cases as termination is undecidable).

For the remainder of this chapter, we consider an arbitrary algorithm $\mathcal{A}$ with n tasks and fixed state and message sets Q and $\mathcal{M}$, together with the corresponding set of configurations C. Further let $b \in \mathbb{N}$ be greater than the maximum number of messages sent by any action in one step.

3.3.1 A termination criterion

Based on the semantic model, we now develop a criterion sufficient for an algorithm to terminate.

First of all, we precisely define what it means for an algorithm to terminate. All considerations are made under the general premise that local actions terminate, i.e., their execution takes a finite amount of time. As further the asynchronous timing model comes with the assumption that messages are delivered after a finite amount of time, each step with the successor relation corresponds to a finite time of execution. We can therefore say that an algorithm (always) terminates on a certain configuration if it can only take finitely many steps from that configuration. Note that because of non-determinism, there exist algorithms where there are finite and infinite executions starting from the same configuration. We only consider the question whether an algorithm *always* terminates and focus on whether this is the case for *all* configurations.

Further note that our conception of termination is in the sense of "there is no more activity, neither locally, nor globally". This does not mean that the algorithm has reached a certain state (e.g. that a protocol has completed) – the execution can also simply get "stuck" or end in a *deadlock*, because tasks are waiting for messages that are not received. For example, the two-phase commit protocol (Algorithm 3.1) would be considered to terminate if all participants have sent their `commit()` vote but the coordinator does not receive all of them and thus cannot continue, resulting in all tasks being blocked, waiting for messages.

Definition 3.6. A distributed algorithm $\mathcal{A}$ is *terminating* on configuration $c_0 \in C$ if every sequence of successor configurations $c_0 \to_{\mathcal{A}} c_1 \to_{\mathcal{A}} \cdots$ is finite. ◁

The idea of the termination criterion is as follows. Assuming that all local actions are terminating, the only way for an algorithm to not terminate is through an infinite amount of messages being sent. Or, more precisely in terms of the semantic model: as each step consumes a message, an infinite sequence of steps requires an infinite amount of messages being produced. As messages are only produced by actions, and actions are only invoked by receiving a message, there must be a "message loop": there must exist a sequence of input-action pairs where each is able to send a message to the next one and the last one can send a message to the first one (with any task configurations). In other words, if we imagine the input-action pairs of an algorithm to form the nodes of a directed graph where there is an edge from x to y if x's action can send a message which can match y's input part, there must be a cycle in this graph. Thus, the existence of a cycle in the described graph is a necessary condition for an algorithm to not terminate. Consequently, if the graph is acyclic, the algorithm is terminating. We now formally define this graph and prove that acyclicity indeed implies termination.

3.3.2 The message flow graph

The *message flow graph* $G(\mathcal{A})$ for an algorithm $\mathcal{A}$ represents the possible control flow between actions, that is, it connects actions if one can be the cause of the other to be executed (which is by a message "flowing" from one to the other). More precisely, a message can "flow" from an action δ to an action δ^* if in a configuration c, δ can produce a message m' (resulting in a configuration c') which can be handled by δ^* in a configuration c^* with $c' \to_{\mathcal{A}}^* c^*$.

We continue to denote by c^* future configurations of a current configuration c (which includes c) and by q^*, δ^* etc. states, actions etc. which can be reached or executed in such a future configuration.

Definition 3.7. The *message flow graph* for $\mathcal{A} = (g_i)_{i \in I}$ is the directed graph $G(\mathcal{A}) = (A, E)$ where for all $\delta, \delta^* \in A, k \in I, c = (q_i, M_i)_{i \in I}, c' = (q_i', M_i')_{i \in I} \in C, m \in M_k$ with $c \to_{\mathcal{A}} c'$ via $(\delta, m) \in g_k(q_k, M_k)$ and $\delta(q_k, m) = (q_k{}', M_1^+, \ldots, M_n^+)$ as in Definition 3.5:

$$(\delta, \delta^*) \in E \iff \exists j \in I, c^* = (q_i^*, M_i^*)_{i \in I} \in C\colon \\ c' \to_{\mathcal{A}}^* c^* \ \wedge \ (\delta^*, m') \in g_j(q_j^*, M_j^+)\ . \qquad \triangleleft$$

Fig. 3.1 shows the message flow graph for the two-phase commit protocol (Algorithm 3.1). The actions are simply labeled by their corresponding guard, prefixed by the task's identifier. Here we have one task C (Coordinator) and n tasks $\mathrm{P}_1, \ldots, \mathrm{P}_n$ (Participant). It is easy to see that the representation of multiple copies of the same task is redundant regarding the existence of cycles in the graph. Therefore, from now on we will only show one representative for a group of identical tasks when visualizing the graph, as done in Fig. 3.2 (that is, here we merge $\mathrm{P}_1, \ldots, \mathrm{P}_n$ into P).

To demonstrate how the reformulation of algorithms by splitting actions into multiple input-action pairs reflects in the message flow graph, Fig. 3.3 shows the message flow graph of Algorithm 3.2.

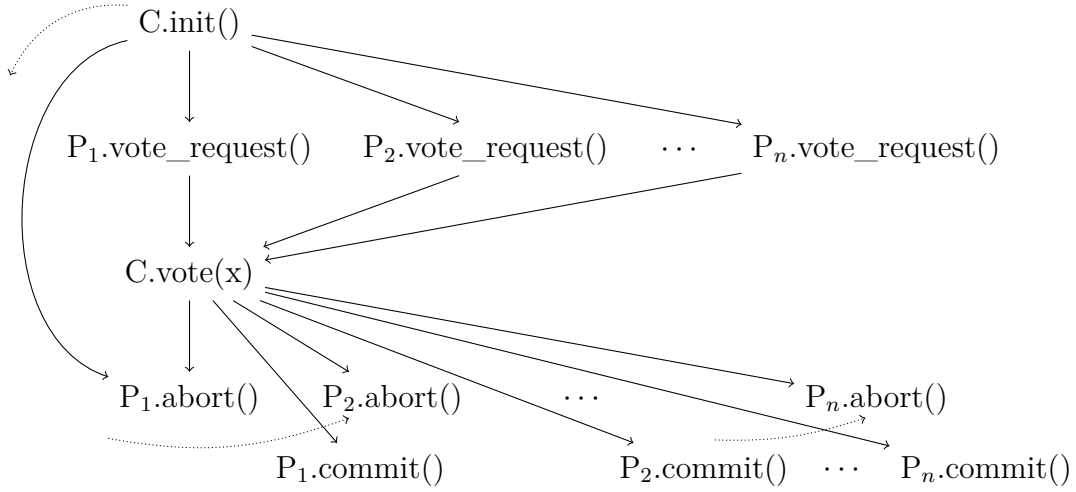


Figure 3.1: The message flow graph of Algorithm 3.1

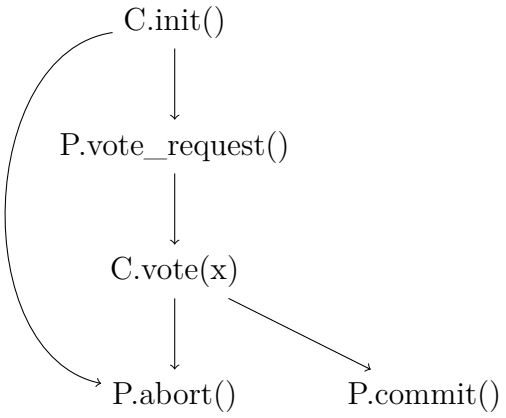


Figure 3.2: The simplified message flow graph of Algorithm 3.1

The message flow graph can be used to visualize an execution of an algorithm. To this end, imagine the graph to be partitioned into one set of nodes per task. As an aside, note that the graph resulting from collapsing the nodes of each of these sets into a single node per task represents the actual communication topology the algorithm uses at most. Now imagine that each time a message is sent, it is associated with those outgoing edges of the sending action which end at the destination task's actions which could currently receive the message, that is, the guard of the corresponding input-action pair is satisfied with that message. A step of the algorithm then consists of moving a message currently associated with at least one edge (it is possible that a message is not associated with any edge) to one of the actions these edges are pointing to. This action is then executed with the chosen message and the newly created messages are associated with edges as described above. The association of messages to edges can change whenever

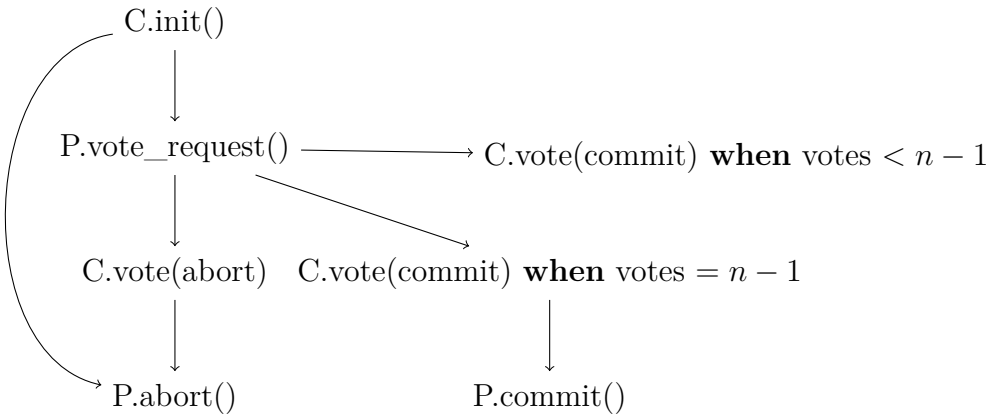


Figure 3.3: The (simplified) message flow graph of Algorithm 3.2

the destination task's state changes. This illustrates the working of guarded commands: depending on the current state and the conditions in the guards, different actions become ready to process a message.

With the definitions of termination and the message flow graph, we can eventually formulate the termination criterion.

Theorem 3.8. If $G(\mathcal{A})$ is acyclic, then $\mathcal{A}$ is terminating on all configurations.

According to this criterion, the two-phase commit protocol (Algorithm 3.1) always terminates, as its message flow graph (Fig. 3.2) is acyclic. Note, however, that in general this criterion is undecidable as the message flow graph is in general incomputable (it is undecidable whether a message can flow from one action to another). After proving correctness of the termination criterion, we show how it can be used in practice anyway.

First, however, we come back to the topic of spontaneous actions. With the formal definition of the message flow graph, it should be clear that if actions could execute spontaneously (with guards based only on the current state, without the need to match and consume a message), we would lose the benefits of terminating actions, that is, to focus on the global behaviour of the algorithm: even if each action terminates, an infinite sequence of actions in a single task, caused by local state changes from the execution of those actions, could make an algorithm not terminate. Whether this can actually be the case is intuitively even more difficult to decide than whether a message can flow from one action to another. With the requirement of a message to be consumed with each action, termination can be analyzed solely based on the communication behaviour represented by the message flow graph.

Before even proving correctness of the termination criterion, we take a look at its inherent limitation: equivalence of acyclicity and termination cannot be shown. An algorithm may terminate on all configurations even though there is a cycle in its message flow graph. This is due to the fact that the message flow graph is based on the local properties between each pair of input-action pairs and has thus no means of expressing global control flow of a sequence of *multiple* activations.

More precisely, the existence of an edge between two actions in the message flow graph means that there exists a configuration where the first action can send a message which, in any future configuration, can be received by the second action. However, this does not mean that for a path $\delta_1 \to \delta_2 \to \delta_3$ in the graph, the invocation of δ_1 can actually cause δ_3 to be invoked: the configurations where a message can be sent from δ_2 to δ_3 could be disjoint from the configurations where δ_2 can receive a message from δ_1. Finally, even if there is a cycle along which actions can successively invoke each other for a whole round, it does not necessarily mean that this is possible infinitely often. These limitations and possible solutions are discussed in Section 3.3.7.

3.3.3 Correctness of the termination criterion

Before formally proving correctness, we intuitively describe the idea of the proof. Assume an initial configuration of an algorithm $\mathcal{A}$ with some messages in the input buffers. According to the message flow graph, messages can only "flow" from certain actions to certain other actions (a message m' can "flow" from δ to δ^* if δ can send m' and δ^* can receive m' at some later point). If the message flow graph is acyclic, then the actions of $\mathcal{A}$ can be ordered from "high" to "low" such that messages only flow from high to low (that is, if an action δ produces a message m', it can only be received by an action lower than δ). As in each step, one message is removed by an action and the finitely many new messages created can only be received by actions lower than the sending action, after finitely many steps the algorithm can take no more steps (either because all input buffers are empty or none of the waiting messages match any of the guards of the corresponding tasks' input-action pairs).

We formally order actions with *ranking functions*:

Definition 3.9. Let $G = (A, E)$ be an acyclic directed graph on actions. A *ranking function* for G is a function $r_G : A \to \mathbb{N}$ with

$$(\delta, \delta^*) \in E \;\Rightarrow\; r_G(\delta) > r_G(\delta^*) \; .$$

We then call $r_G(\delta)$ the *rank* of δ. ◁

Note that ranks do not have to be unique. The definition only requires the ranks of actions in the same strongly connected component to be unique which is sufficient for our purposes.

To formalize the proof idea, we assign values to messages based on the ranks of actions that can receive the message. Then, if messages only "flow from high to low", the values of messages subsequently created as the cause of each other will decrease.

To begin with, the value of a message m (in input buffer M_k) in a certain configuration is the maximum rank of an action (of task k) which can handle m in the current or any future configuration.

Definition 3.10. For a ranking function r_G, a configuration $c = (q_i, M_i)_{i \in I} \in C$ and $k \in I$ we define the *message value* of $m \in M_k$ by

$$y_k^{r_G}(c, m) = \max\{\, r_G(\delta^*) \mid \exists c^* = (q_i^*, M_i^*)_{i \in I} \in C : \\ c \to_{\mathcal{A}}^* c^* \ \wedge\ (\delta^*, m) \in g_k(q_k^*, M_k^*) \,\}$$

where we define $\max(\emptyset) = 0$. ◁

The next two lemmas are the key part in showing that an algorithm with an acyclic message flow graph will terminate. The first one shows that if a message m causes a message m' to be produced, the value of m' is lower than that of m.

Lemma 3.11. Let the message flow graph $G(\mathcal{A}) = (A, E)$ be acyclic with ranking function r_G. Further let $k \in I, c = (q_i, M_i)_{i \in I}, c' = (q_i', M_i')_{i \in I} \in C, m \in M_k$ with $c \to_{\mathcal{A}} c'$ via $(\delta, m) \in g_k(q_k, M_k)$ and $\delta(q_k, m) = (q_k', M_1^+, \ldots, M_n^+)$ as in Definition 3.5. Then for all $j \in I$ and $m' \in M_j^+$ we have $y_k^{r_G}(c, m) > y_j^{r_G}(c', m')$.

Proof. Let $j \in I$ and $m' \in M_j^+$. For all $c^* = (q_i^*, M_i^*)_{i \in I} \in C$ with $c' \to_{\mathcal{A}} c^*$ and for all $\delta^* \in A$ with $(\delta^*, m') \in g_j(q_j^*, M_j^*)$ we have $(\delta, \delta^*) \in E$ by the definition of the message flow graph and thus $r_G(\delta) > r_G(\delta^*)$. It follows:

$$\begin{aligned} r_G(\delta) &> \max\{\, r_G(\delta^*) \mid \exists c^* = (q_i^*, M_i^*)_{i \in I} \in C : \\ &\qquad\qquad c' \to_{\mathcal{A}}^* c^* \ \wedge\ (\delta^*, m') \in g_j(q_j^*, M_j^*) \,\} \\ &= y_j^{r_G}(c', m') \ . \end{aligned}$$

With

$$\begin{aligned} y_k^{r_G}(c, m) &= \max\{\, r_G(\delta^*) \mid \exists c^* = (q_i^*, M_i^*)_{i \in I} \in C : \\ &\qquad\qquad c \to_{\mathcal{A}}^* c^* \ \wedge\ (\delta^*, m) \in g_k(q_k^*, M_k^*) \,\} \\ &\geq r_G(\delta) \end{aligned}$$

(as for $c^* = c$, by premise δ is one of the δ^* in the set) we obtain

$$y_k^{r_G}(c, m) > y_j^{r_G}(c', m')$$

as desired. □

Now we define the value of an algorithm's configuration to be simply the sum of all message values of the messages in the input buffers, each as a power of b (the constant we defined to be greater than the maximum number of messages sent in a step). The exponentiation ensures that up to $b - 1$ messages of a lower value than that of a message m together contribute less to the sum than m, as $(b-1)b^{x-1} < b^x$ for all $x \geq 1$.

Definition 3.12. For a ranking function r_G, the *configuration value* of a configuration $c = (q_i, M_i)_{i \in I} \in C$ is

$$v_{r_G}(c) = \sum_{i \in I} \sum_{m \in M_i} b^{y_i^{r_G}(c, m)} \ . \qquad \triangleleft$$

The following lemma lifts Lemma 3.11 from message values to configuration values and completes the requirements to prove termination. It shows that the value of a configuration can only decrease when an algorithm with an acyclic message flow graph takes a step. Here we finally make use of the requirement that in each step a message is consumed. The removed message decreases the configuration value and by Lemma 3.11 the values of the newly created messages are lower than that of the removed, which together with the bound b on the number of created messages results in an overall decrease of the configuration value.

Lemma 3.13. Let $G(\mathcal{A})$ be acyclic with ranking function r_G. Let $c, c' \in C$ with $c \to_{\mathcal{A}} c'$. Then $v_{r_G}(c) > v_{r_G}(c')$.

Proof. For $c = (q_i, M_i)_{i \in I}$ and $c' = (q'_i, M'_i)_{i \in I}$ let $k \in I, m \in M_k$ and $M_1^+, \ldots, M_n^+$ as in Definition 3.5. Then

$$v_{r_G}(c') = v_{r_G}(c) - b^{y_k^{r_G}(c,m)} + \sum_{j \in I} \sum_{m' \in M_j^+} b^{y_j^{r_G}(c',m')} .$$

With $\sum_{j \in I} |M_j^+| < b$ and, by Lemma 3.11, $y_k^{r_G}(c,m) > y_j^{r_G}(c',m')$ for all $j \in I$ and $m' \in M_j^+$ and thus $b^{y_k^{r_G}(c,m)} > (b-1)b^{y_j^{r_G}(c',m')}$ (for any $j \in I$ and $m' \in M_j^+$) we obtain the claimed inequality. □

Finally, we can prove Theorem 3.8.

Proof (Theorem 3.8). Let $G(\mathcal{A})$ be acyclic with ranking function r_G and let $c_0 \in C$ be a configuration. Let $c_0 \to_{\mathcal{A}} c_1 \to_{\mathcal{A}} \cdots \to_{\mathcal{A}} c_\ell$ be any sequence of successor configurations starting from c_0. Then by Lemma 3.13 $v_{r_G}(c_0) > v_{r_G}(c_1) > \cdots > v_{r_G}(c_\ell)$. As $v_{r_G}(c) \geq 0$ for all $c \in C$, we obtain $\ell \leq v_{r_G}(c_0)$. Thus, every sequence of successor configurations must be finite, i.e., $\mathcal{A}$ is terminating. □

We have shown that algorithms with acyclic message flow graphs always terminate on every configuration. Without proof (though intuitively clear, the proof would be rather technical), we generalize Theorem 3.8 to consider termination for a single configuration: when a certain initial configuration is of interest, only the part of the message flow graph reachable from that configuration has to be considered.

Theorem 3.14. Let $c = (q_i, M_i)_{i \in I} \in C$ be a configuration and let

$$\begin{aligned} A_c = \{\, \delta^* \in A \mid\ & \exists c^* = (q_i^*, M_i^*)_{i \in I} \colon c \to_{\mathcal{A}}^* c^* \\ & \wedge\ \exists k \in I, m \in \mathcal{M} \colon (\delta^*, m) \in g_k(q_k^*, M_k^*) \,\} \end{aligned}$$

be the set of actions that can be executed in c or any future configuration. Let G_c be the graph of nodes and edges reachable from A_c in $G(\mathcal{A})$. Then, if G_c is acyclic, $\mathcal{A}$ is terminating on c. □

In practice, however, it is unlikely that the message flow graph contains cycles not reachable from initial configurations.

3.3.4 Spontaneous actions revisited

Recall that we disallowed spontaneous actions (actions initiated without the consumption of a message) so that we could establish a termination criterion based on the global behaviour of an algorithm. We now show how spontaneous actions can still be simulated within the model and how their usage makes the termination criterion useless. We then show a way to bypass this problem, allowing spontaneous actions if they are triggered from outside the algorithm. Termination can then be seen relative to such external invocations.

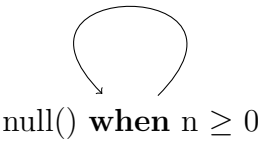
3.3.4.1 Simulating spontaneous actions

While spontaneous actions are not directly supported by the model, it is still possible to simulate them. What prevents true spontaneous actions is the fact that for the execution of an action, a message must be consumed. To effectively allow spontaneous actions, it suffices to have a special dummy message in each input buffer at all times. This message can be received by any input-action pair to execute a spontaneous action. The action simply has to restore this message by sending it to its own task so that further spontaneous actions are possible. With respect to the termination analysis, this restoration is the problem: an action sending a message it can receive itself creates a loop in the message flow graph. Algorithm 3.4 shows such a situation with the corresponding message flow graph (`null()` is the dummy message).

Spontaneous: { S }

Variables
    n := 0

input null() **when** n ≥ 0 {
    n++
    send null() **to** S
}



Algorithm 3.4: Simulating spontaneous actions (with the corresponding message flow graph)

The described technique of sending messages to oneself can actually be useful to perform repeated actions. If it is known that a particular application of this technique cannot lead to infinite executions and input-action pairs are constructed in an appropriate way, cycles in the message flow graph can actually be avoided. The additional techniques required to do so are discussed in Section 3.3.7.3. Next we want to take a look at a possibility of allowing spontaneous actions in a way compatible with the termination analysis.

3.3.4.2 External messages as an alternative

A different way of simulating spontaneous actions is by explicitly allowing each such action externally. The technique with dummy messages can be used as just

described, only that the messages are not created (restored) by the algorithm itself but added externally. To this end, the configuration of an algorithm is modified at any point during its execution by adding dummy messages to the input buffers. The advantage of this concept is that no cycles are added to the message flow graph and the occurrence of spontaneous actions can (but also has to) be controlled externally.

With external messages, termination can now be seen relative to each external message. Adding an external message to a configuration creates a new configuration (of a higher value) but Theorem 3.8 still applies: from that new configuration, the algorithm can only take finitely many steps (if the message flow graph is acyclic). Thus, assuming that only finitely many external messages are added, an algorithm with an acyclic message flow graph is still going to terminate.

Instead of adding dummy messages at runtime, an algorithm can be provided with a one-time budget of dummy messages in its initial configuration – then the termination criterion even applies exactly as before, for the whole execution. This has actually practical applications: it allows to repeat a certain protocol for a fixed number of times, which is for example used to increase fault tolerance.

3.3.5 Applying the termination criterion

There is one missing part in turning the termination criterion into a termination analysis: we have not discussed yet how the message flow graph can be constructed in practice. Actually, as it is a semantical and thus undecidable property whether a message can flow from one action to another, the message flow graph can in general not be constructed. Therefore, the goal of this section is to overapproximate it in a practical way. To start with, we generalize Theorem 3.8 for graphs overapproximating (that is, containing) the message flow graph.

Corollary 3.15. Let G be an acyclic directed graph that contains $G(\mathcal{A})$. Then $\mathcal{A}$ is terminating on all configurations.

Proof. If G is acyclic and contains $G(\mathcal{A})$, $G(\mathcal{A})$ is also acyclic. By Theorem 3.8, $\mathcal{A}$ is then terminating on all configurations. □

In the following we discuss how the message flow graph can be approximated.

3.3.6 Approximating the message flow graph

Approximations with different levels of granularity can be identified for the message flow graph. A very coarse approximation is the topology graph of an algorithm, the directed graph on tasks where an edge represents the general possibility of a message being sent from one task to another (as given by a physical or logical network topology). This graph can be turned into a graph on actions by connecting each action of a task A to all actions of a task B if there is an edge from A to B in the topology graph. Of course this approximation is so rough that it is not very relevant to practical applications.

To differentiate between the individual actions of a task, we have to go a little deeper into the semantics. The graph connecting only those actions where one can actually produce a message which can later be received by the other is already the incomputable message flow graph. The goal is thus to find something in between. We need a reasonable and decidable criterion to have an edge between two actions which is always true if the respective edge is present in the message flow graph.

The precision of the criterion regarding this implication (how close it comes to equivalence) determines how close the resulting graph is to the message flow graph and thus determines the precision of the termination analysis based on message flow graphs. In the end, a balance has to be found between precision on the one side and practicability and computational complexity on the other side.

A way to obtain a decidable criterion for an edge in a graph that should approximate the message flow graph is to base it on syntactic properties of an algorithm's textual representation which imply the desired semantic properties. Whereas until now our considerations about termination have only been based on the semantic model, from now on we assume algorithms to have a corresponding program text with the syntax from Section 3.2.4 (which we are going to refine shortly).

3.3.6.1 The message type graph

A quite natural and efficiently computable syntactic property which can be used to approximate the message flow graph can be obtained with the use of *message types*.

Note that until now we have not specified any structure for messages, a message is simply an element from the unstructured set $\mathcal{M}$. Now we require each message to be of a previously defined type. The purpose is to narrow down the usually infinite amount of possible messages to a finite amount of syntactically identifiable message types. A way of integrating message types into the input-action language, which at the same time is useful when formulating algorithms, is through *message signatures* in the form of type signatures. For example, the signature of a message requesting from a server an item at a certain position of a named list could look like `get_item(String list, int pos)`.

We have actually used this kind of message types in Algorithm 3.3 and also our other examples use message types, just that their types only consist of the message name and not the parameter list. For the concept of message types it makes no difference whether each type has to have a different name or the list of parameter types belongs to the type's identity (and thus allows for "message type overloading"). In the following, however, we assume each message type to have a unique name and be associated with a certain parameter list.

To be a bit more precise, we refine the syntax introduced in Section 3.2.4 to include message types. To this end, $\langle msg \rangle$ is refined to specify a message type and a list of values for the corresponding parameters. Similarly, the guard of an input-action pair must specify the message type to accept plus a list of variables to be assigned the values of the matched message. Note that while in general one can think of more sophisticated message matchers (like pattern matching

which can also make use of task variables), for simplicity here we move all further conditions on the message to the `when` clause.

⟨*task*⟩ ::= '`task`' ⟨*task-identifier*⟩ '`{`'
⟨*variable-declarations*⟩
{ ⟨*input-action-pair*⟩ }
'`}`'

⟨*input-action-pair*⟩ ::= ⟨*input*⟩ ⟨*action*⟩

⟨*input*⟩ ::= '`input`' ⟨*msg-type*⟩ ⟨*var-list*⟩ ['`when`' ⟨*boolean-exp*⟩]

⟨*action*⟩ ::= '`{`' { ⟨*statement*⟩ } '`}`'

⟨*statement*⟩ ::= '`send`' ⟨*msg*⟩ '`to`' ⟨*dest*⟩
| '`reply`' ⟨*msg*⟩
| ...

⟨*msg*⟩ ::= ⟨*msg-type*⟩ ⟨*value-list*⟩

Based on a program text with message types we can now construct a graph on actions matching message types in actions' `send` and `reply` statements with message types in actions' guards, ignoring possible conditions for the `send` or `reply` statement to be executed, actually possible receivers according to the `send` statement's `to` part as well as additional conditions in the guards. We call this graph the *message type graph* for $\mathcal{A}$. The definition of the message type graph must stay somewhat informal, as we have not fully formalized the syntax and its translation into the semantic model.

Definition 3.16. The *message type graph* for an algorithm $\mathcal{A}$ (given in a syntax using message types) is the directed graph $G_T(\mathcal{A}) = (A, E)$ where $(\delta, \delta^*) \in E$ iff δ contains a `send` or `reply` statement with the message type specified in the guard of δ^*. ◁

It is clear that the criterion for an edge in the message flow graph (the possibility of a message to flow along the edge) implies the criterion for the same edge in the message type graph (the mere matching of message types). Therefore, an algorithm's message type graph contains its message flow graph.

Lemma 3.17. $G_T(\mathcal{A})$ contains $G(\mathcal{A})$. □

With Corollary 3.15, we can formulate the termination criterion in terms of message type graphs.

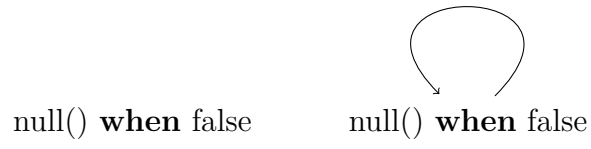
Corollary 3.18. If $G_T(\mathcal{A})$ is acyclic, then $\mathcal{A}$ is terminating on all configurations. □

3.3.6.2 Precision of the message type graph

We notice that for some algorithms, as for example the two-phase commit protocol, the message type graph is already precise enough to infer termination. For Algorithm 3.1, the message type graph is actually identical to the message flow graph (Fig. 3.2). The reason is that each message flow included in the message type graph is actually possible. This is indeed the case for all (real) algorithms presented in this work.

For illustration, we now also construct examples where the message type graph differs from the message flow graph. Algorithm 3.5 consists of a repeated action triggering itself where the corresponding guard has an unsatisfiable condition. Therefore, no edge can point to this input-action pair in the message flow graph. The message type graph, however, has a loop as it does not consider additional conditions in guards.

```
Impossible: { S }
input null() when false {
    send null() to S
}
```



Algorithm 3.5: Impossible message flow – the message flow graph (middle) has no edges, the message type graph (right) has a loop

This example was very hypothetical and we want to take a look at another, slightly more realistic example. Here we want to consider the message flow graph restricted to flows actually reachable from an initial configuration as defined in Theorem 3.14 (here the initial state is given by the initial variable values in the algorithm). Algorithm 3.6 consists of two tasks where each waits for the other to send a message but for receiving this messages they rely on the earlier reception of such a message. Thus, while there is a task state where the reception is possible, this state is not reachable from the initial configuration. In the message flow graph there are hence no edges (Fig. 3.4) but the message type graph again contains a cycle (Fig. 3.5).

```
Blocked A: { A }
Variables
    b := false
input b() when b {
    b := true
    send a() to B
}
```

```
Blocked B: { B }
Variables
    a := false
input a() when a {
    a := true
    send b() to A
}
```

Algorithm 3.6: Non-executable actions result in differences of message flow graph and message type graph.

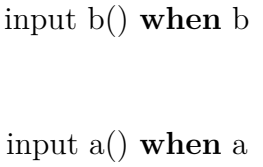


Figure 3.4: The message flow graph of Algorithm 3.6 restricted to reachable configurations as in Theorem 3.14

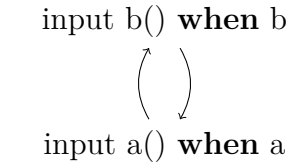


Figure 3.5: The message type graph of Algorithm 3.6

3.3.6.3 Better approximations

In the previous section, we have seen examples where edges present in the message type graph are not present in the message flow graph. One way of achieving a better approximation of the message flow graph would thus consist of determining for which edges this is the case.

We can identify multiple reasons why a message flow in the message type graph is actually not possible. However, in general this can depend on the semantics of the whole algorithm which is why it is not obvious whether the message type graph differs from the message flow graph. Apart from unsatisfiable `when` clauses, impossible flows can arise from `send` statements which can actually never be executed or `to` parts of `send` statements restricting possible receivers. It gets more complex when conditions on both sides, sender and receiver, are concerned: in general, it might be possible for a message to be sent and received but there is no intersection in possible message content sent and message content satisfying the corresponding input condition.

To develop static analyses based on the above aspects, standard techniques for sequential program analysis can be used. However, it can be argued that the described opportunities for better approximation are not that relevant in practice: if a programmer adds combinations of `send` statements and guards which do not allow a message flow but still result in an edge in the message type graph, it can be considered a design mistake. The message type graph includes all potential message flows the designer has explicitly specified according to the message types, whether they are actually possible or not. Different message types can be used to avoid situations where a certain flow is wanted and another, yet included in the message type graph, is not. Turning the argumentation around, however, these analyses can be used to reveal design mistakes.

3.3.6.4 Practicability of the message type graph

The message type graph can be constructed very efficiently: to match message types in `send` statements with those in guards, after identifying all occurrences of `send` statements and their message types in one scan of the program code, it suffices to compare each pair of input-action pairs once. The construction can thus be done in quadratic time w.r.t. the number of input-action pairs. Further, a graph can be checked to be acyclic in quadratic time in the number of its nodes. Therefore, termination based on Corollary 3.18 can be decided in quadratic time

based on the number of input-action pairs (excluding parsing).

We note that apart from serving as the basis of the termination analysis, the message type graph has a further practical use case: already during the design process it allows the designer to visualize all potential message flow. When an algorithm gets larger, it gets surprisingly difficult to reliably identify all possible connections between input-action pairs by hand. Thus, the message type graph helps in verifying that the algorithm contains exactly the intended potential message flows.

While there certainly is a gap between the message type graph and the message flow graph, we have argued that in practice, the message type graph is already a quite good approximation. Regarding how efficiently it can be constructed, we adopt it as an appropriate choice for general use.

3.3.7 Improving precision

Obviously, the precision of the termination analysis depends on the quality of the approximation of the message flow graph. However, as argued in the previous section, the gap between the practical message type graph and the theoretic message flow graph is not that big or even non-existent in practical applications. In this section, as a prospect on future work, we want to explore other ways of making the termination analysis more precise than just better approximating the message flow graph.

The inherent imprecision of a termination analysis based on message flow graphs is that cycles in the message flow graph do not imply non-termination (cf. Section 3.3.2). We present two approaches of getting around this problem. The first is about identifying cycles which actually cannot even be traversed as a whole. The second is about identifying cycles which can only be traversed finitely often and a way of refining the message flow graph by adding more nodes, with the goal to get rid of such cycles. To start with, we take a closer look at the limitations of the message flow graph.

3.3.7.1 Limitations of the message flow graph

The message flow graph's imprecision is a matter of global control flow. Even if its definition might give the impression (a configuration where a message is received must be reachable from a configuration where the message is sent), the message flow graph does not really consider sequences of configurations in an algorithm's execution.

First of all, recall from Section 3.3.2 that a path $\delta_1 \to \delta_2 \to \delta_3$ in the message flow graph does not mean the invocation of δ_1 can actually cause δ_3 to be invoked. The presence of an edge between two actions is a local property without a context of global control flow.

Then, it is not given that the configuration serving as the starting point for a message flow represented by an edge is even a realistic one (i.e., reachable from an initial configuration). This is what Theorem 3.14 tries to address by restricting the message flow graph to actions reachable from an initial configuration. However, this still considers each action separately and not actual executions: a certain

action might be executable when reaching it on a certain path from the initial configuration, another when reaching it via another path.

3.3.7.2 Harmless cycles

A cycle in the message flow graph does not imply non-termination. Some cycles might not even be a *potential* source of non-termination as they cannot be traversed as a whole, i.e., the subsequent activation of all actions on the cycle is not possible. We now present an idea on how such "harmless" cycles can be identified. In the process, we address the two limitations of message flow graphs mentioned in the previous section.

To see whether the successive activation ("traversal") of a whole path of actions is possible, possible sending and receiving configurations for each pair of subsequent actions on the path have to be identified. Given a path $\delta_1 \to \delta_2 \to \delta_3 \to \cdots \to \delta_\ell$, we would identify sets of configurations $C_{1/2}, C_{2/3}, \ldots, C_{\ell-1/\ell}$ where $C_{i/i+1}$ represents the configurations where δ_i can send a message which in a future configuration can be received by δ_{i+1}. As these sets are in general not computable, they must be overapproximated – in the worst case they include all configurations C. An approximation could start with C and use specific local analyses which eliminate certain configurations. The idea is now to connect these sets: the configurations in $C_{2/3}$ have to be reachable from at least one configuration in $C_{1/2}$ and so on. We will not go into detail how this can be achieved. Depending on the static analyses used on local actions, global as well as iterative approaches might be of consideration.

The goal is to eventually find one set $C_{i/i+1}$ on the path which is empty, meaning that there is no execution of the algorithm arising from a sequence of actions including the actions on the given path. A flow along all the length of the path is then not possible and a cycle containing this path is no potential source of non-termination. Note that this only holds for cycles including the exact path – parts of this path may still be part of other paths which actually represent a possible flow. This is why the message flow graph cannot simply be altered by removing edges of such paths and each cycle has to be considered separately.

To address the second limitation of the message flow graph, that for possible message flow only configurations reachable from certain initial configurations should be considered, the configurations serving as starting point for the above sequences of actions can be restricted. To this end, the set $C_{1/2}$ can be narrowed down to configurations reachable from initial configurations. The result can then propagate to other sets $C_{i/i+1}$.

To actually apply the idea, apart from formally identifying harmless cycles (where because of undecidability not all can be found) the termination criterion would have to be adapted to account for harmless cycles in the message flow graph which is probably not trivial.

An alternative is using heuristics to find out whether a cycle can actually be traversed completely. If such a cycle is found, the likelihood that the algorithm might not terminate can be considered higher (it is still possible that the algorithm always terminates) and the more of the search space has been explored without finding a cycle, the lower this likelihood can be considered. However, as the search

space is usually infinite, the procedure might not terminate, or an error-probability has to be accepted when aborting after a certain coverage.

3.3.7.3 Limited edges

Our second approach to improve precision is about identifying special edges in the message flow graph which allow us to change its structure, with the goal of removing cycles.

Cycles in the message flow graph represent a potential source of non-termination as they could allow an infinite message flow around the cycle. The goal of the previous section was to find out whether the whole cycle can be traversed at all. Here the question is whether an infinite traversal is possible. If a cycle contains an edge that can only be traversed finitely often, this cycle does no more represent a possibility of non-termination. To make use of this observation, two steps are required: the identification of edges in the message flow graph which can only be "traversed" finitely often (which in general is undecidable) and a way to infer termination if all cycles are "broken" by such a *limited edge*.

We start with showing that with the knowledge of limited edges, a more precise termination analysis can be obtained. First, we take an intuitive look at what limited edges mean for the message flow in the message flow graph before finding a syntactic transformation with the desired effect of removing cycles.

Recall how a message flow graph can be used to visualize the execution of an algorithm by actually letting messages flow along its edges. The idea is now that if the traversal of an edge in a cycle is limited by a certain number, in an actual execution the replacement of that cycle with a finite path (with copies of edges accounting for the maximum number of traversals) cannot be noticed. Cycles with limited edges can thus be "unrolled" by putting copies of their nodes (including all incoming and outgoing edges) repeatedly next to each other. The unrolled graph can be obtained from the original graph using a graph traversal algorithm (e.g. DFS) which traverses edges as often as their given limit permits, each time creating a new copy of the passed nodes and edges. Unbounded cycles have to be detected and added to the result. The resulting graph still represents all possible message flow the original message flow graph includes and can thus intuitively be used to apply the termination criterion.

To actually apply the idea and obtain such an unrolled message flow graph, we need a syntactic transformation of the algorithm which corresponds to the unrolling of its message flow graph while preserving its original semantics. The idea is to create copies of input-action pairs corresponding to the copies created while unrolling. This includes modifying messages to be sent so that they can only reach the input-action pairs according to the unrolled message flow graph. We do this by introducing new message types. That is, edges to different copies of input-action pairs in the unrolled message flow graph also correspond to different message types being sent. This ensures that the same unrolling effect is achieved in the message type graph which is required to practically make use of the transformation.

To fully implement the idea, we would have to describe a precise unrolling algorithm considering all special cases and then prove that the resulting syntactic

transformation indeed preserves semantics. As this is beyond the scope of this work, we leave it at the idea. Instead, in the next section we apply the idea to a concrete algorithm to demonstrate its working.

However, before doing so we come back to the second requirement to make use of limited edges, namely their identification. There are different reasons why a certain message flow can only happen for a limited number of times: the limit can have its origin in the context of the corresponding `send` statements, the guards of input-action pairs or the content of the messages. The detection is of course undecidable in general. However, if for example the invalidation of a guard after a certain number of activations is evident, already unsophisticated analyses can detect some limited edges. Another possibility is providing special syntactic elements which facilitate the detection of limited edges. A trivial form of such a syntactic element is an annotation claiming that a certain input-action pair will only be activated for a certain number of times. The claim's correctness can also be enforced by the runtime system by counting activations and transiting into an error state should the given number be exceeded. A more sophisticated syntactic support would address certain algorithmic design patterns that usually result in cycles in the message flow graph.

3.3.7.4 Inferring termination for a ring algorithm

A class of algorithms where cycles in the message flow graph are typical are ring algorithms. Their characteristic is that tasks can only send messages to the next task on the ring. This results in patterns where often a single message is passed around the whole ring. The tasks on the ring usually have identical functionality and the passing around of a message is handled by a single input-action pair by receiving, processing and relaying it until a certain condition is met. If a message is supposed to travel a whole round, there will be a cycle in the message flow graph. To avoid this cycle using the idea of limited edges, the relaying input-action pair can be marked with a known limit of maximal traversals.

We demonstrate the idea of syntactically specifying limited edges and the resulting transformation based on an unrolled message flow graph with a ring algorithm for *leader election*. The leader election problem consists of finding a single task to be the leader of a group, which should be known to all tasks. Leaders (or coordinators) are an essential component of many algorithms, as for example in the two-phase commit protocol. Leader election usually takes place when a task detects that the current leader is not responding anymore and is thus assumed to have failed.

The *Chang and Roberts algorithm* (Algorithm 3.7) [CR79] implements leader election on a ring topology. It assumes that each task has a unique ID with a total order on IDs. The idea is to make the task with the highest ID be leader. The protocol proceeds in two phases: an election phase and an announcement phase. The election phase begins with an initiating task sending an `elect(i)` message with its own ID to its neighbour. Each task receiving an `elect(i)` passes on the maximum of the received and its own ID to the next task. This stops when a task receives its own ID, as in this case it must be the task with the highest ID. This task then starts the announcement phase by putting a `leader(i)` message

with its own ID onto the ring which is relayed by each task until the initiating task again receives its own ID.

Obviously, the message flow graph contains cycles, for both, the `elect(i)` and `leader(i)` input-action pairs. Still, it is easy to see that the first message can reach each task at most twice and the second at most once. Specifying these limits syntactically with the `limit` keyword results in some limited edges: with the activations of the `elect(i)` and `leader(i)` input-action pairs being limited to 2 and 1, we obtain global limits for the cycles of $2n$ and n (note that the syntactically specified limits are *per task* whereas the message flow graph represents *global* control flow). Unrolling the message flow graph yields an acyclic graph (Fig. 3.6). The corresponding syntactic transformation of Algorithm 3.7 results in Algorithm 3.8. It is easy to see that both algorithms are semantically equivalent: having accepted that the limits are correct, it is clear that the case where $\texttt{elect}_{2n}\texttt{(i)}$ or $\texttt{leader}_{n}\texttt{(i)}$ would send another `elect(i)` or `leader(i)` message cannot occur (which is why in the code the corresponding `send` statements are replaced by `error` statements). As Fig. 3.6 is actually the message flow graph of Algorithm 3.8, with the transformation based on limited edges we were able to show that the Chang and Roberts ring algorithm for leader election always terminates.

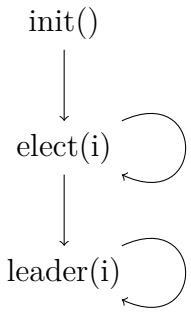
Ring node

```
Variables
    leader // the current leader
    self // own ID
    next // the next task on the ring

input init() {
    send elect(self) to next
}

input elect(i) limit 2 {
    if i = self then
        send leader(i) to next
    else
        send elect(max(i, self)) to next
    end
}

input leader(i) limit 1 {
    leader := i
    if i ≠ self then
        send leader(i) to next
    end
}
```



Algorithm 3.7: The Chang and Roberts ring algorithm for leader election with the corresponding message flow graph

As cycling messages are such a fundamental pattern of ring algorithms, one can also think about introducing special syntax for such messages, making the explicit declaration of limits obsolete. The observation making this possible is that the criterion stopping the cycling is often quite simple. In general it should not be difficult to find a function on the cycling message and/or the current task state which at each relaying is increased at least by a given minimum value and for which the exceeding of a certain value implies the abortion criterion for relaying. With this function and the guarantee of its increasing built into syntax, limited edges could be statically inferred.

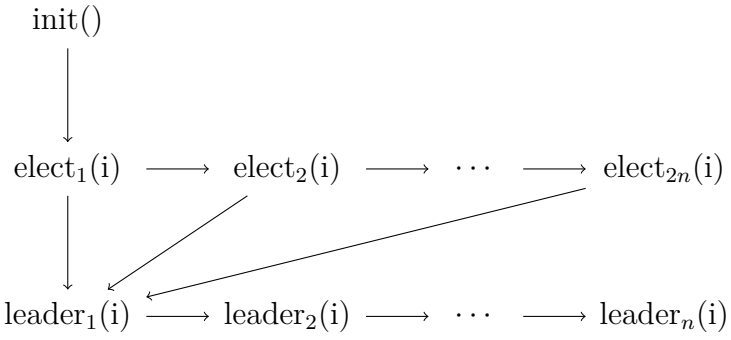


Figure 3.6: The unrolled message flow graph of Algorithm 3.7

To conclude, with additional knowledge we were able to refine the message flow graph by unrolling cycles into finite paths. We also sketched a syntactic transformation for algorithms to obtain refined message flow and message type graphs. This way, the same termination criterion results in a more precise termination analysis.

3.4 Conclusion

In this chapter, we have motivated the input-action language as a suitable language for the specification of distributed algorithms with regard to termination analysis. We have developed a semantic model for the language and based on that model a termination analysis, the correctness of which we have shown.

The termination analysis is founded on the event-driven nature of the input-action language. Entry points for actions in each task are precisely specified by the guards of the input-action pairs. This made it possible to establish a global concept of control flow, the *message flow graph*. To focus on an algorithm's global behaviour regarding termination, we had to make the restriction that tasks can only perform computation on the reception of messages, that is, disallow spontaneous actions. Given this, acyclicity of the message flow graph implies termination but cycles do not imply non-termination as edges represent possible message flow only based on local properties and sequences of activations are not considered.

Ring node

Variables
leader *// the current leader*
self *// own ID*
next *// the next task on the ring*

input init() {
send elect_1(self) **to** next
}

input elect_1(i) {
if i = self **then**
send leader_1(i) **to** next
else
send elect_2(max(i, self)) **to** next
end
}

input elect_2(i) {
if i = self **then**
send leader_1(i) **to** next
else
send elect_3(max(i, self)) **to** next
end
}

...

input elect_{2n}(i) {
if i = self **then**
send leader_1(i) **to** next
else
error "Limit exceeded"
end
}

input leader_1(i) {
leader := i
if i ≠ self **then**
send leader_2(i) **to** next
end
}

input leader_2(i) {
leader := i
if i ≠ self **then**
send leader_3(i) **to** next
end
}

...

input leader_n(i) {
leader := i
if i ≠ self **then**
error "Limit exceeded"
end
}

Algorithm 3.8: The Chang and Roberts ring algorithm for leader election, transformed according to the unrolled message flow graph (Fig. 3.6)

As the message flow graph with its semantic definition is in general not computable, we introduced the *message type graph*, an efficiently computable syntactic approximation of the message flow graph which we think is an adequate choice in practice. The message type graph allows for example to statically infer termination of the two-phase commit protocol. Here the message type graph is identical to the message flow graph and we argued that this probably holds for most practical applications if algorithms are designed reasonably. In addition, the message type graph supports the design process by visualizing potential message flows.

Finally, addressing the limitation of the message flow graph that cycles do not imply non-termination, we sketched two ways of making the termination analysis more precise. The first consists of identifying cycles which cannot be traversed as a whole. The second makes use of limited edges, edges which can only be traversed finitely often. Ring algorithms, where cycles typically occur in the message flow graph but with edges usually being limited, especially profit from this technique. Using the idea of limited edges, we were able to show that the Chang and Roberts ring algorithm for leader election always terminates.

In a nutshell, we have shown that the input-action language lends itself to termination analysis, by developing a termination criterion with the potential of further improvement.

CHAPTER 4

Expressivity

An important field of non-functional properties is fault tolerance, one aspect of which is dealing with expected but not arriving messages. A first step towards analyzing how fault tolerant an algorithm is is the linguistic ability to express the recognition of faults. In this chapter, as an extension to the input-action language, we introduce timers as a way of dealing with the absence of expected messages in the asynchronous timing model. We show that the extension is compatible with the termination analysis described in the previous chapter.

4.1 Timers

The loss of a message or the failure of a task does not necessarily mean that an algorithm will fail. There are different ways of recovering from failures, among them the explicit request to a task to resend a message expected but not received or excluding a task from further participation in a protocol. In any case, before being able to deal with a failure, it has to be detected. Note that (the effects of) task failures can only be noticed via communication. As the asynchronous timing model makes no assumptions about message delivery time, one can never tell whether a message has been lost, has not been sent because of a task failure or is still on the way. In practice, one uses timeouts as an approximation: if an expected message has not arrived after a certain amount of time, it is assumed to never arrive.

For now, the input-action language has no direct means of expressing the waiting for a certain amount of time. To make this possible, we introduce linguistic support allowing to declare, start and stop timers as well as to associate actions with timers, to be executed on a timeout.

We note that there is a use case for timers apart from failures: algorithms might not want to or need to wait for each possible message – sometimes it is part of the specification that a task only waits for a certain amount of time for requests or replies and then continues with its protocol.

4.1.1 Syntax

We add the following new constructs to the language. A timer must be declared in the ⟨*timer-declarations*⟩ part of a task, where it is assigned a unique identifier. Then, the new statements '`start`' and '`stop`' can be used to start and stop a certain timer. Finally, there is a new task element: timeout actions. A timeout

action allows to associate an action (a sequence of statements) with a declared timer. We leave open the concrete form of timer declarations and identifiers.

The following EBNF grammar describes the extended input-action language, based on the syntax from Section 3.2.4.

⟨*task*⟩ ::= '`task`' ⟨*task-identifier*⟩ '`{`'
⟨*variable-declarations*⟩
⟨*timer-declarations*⟩
{ ⟨*task-element*⟩ }
'`}`'

⟨*task-element*⟩ ::= ⟨*input-action-pair*⟩
| ⟨*timeout-action*⟩

⟨*input-action-pair*⟩ ::= ⟨*input*⟩ ⟨*action*⟩

⟨*timeout-action*⟩ ::= '`timeout`' ⟨*timer-id*⟩ ⟨*action*⟩

⟨*input*⟩ ::= '`input`' ⟨*msg-matcher*⟩ ['`when`' ⟨*boolean-exp*⟩]

⟨*action*⟩ ::= '`{`' { ⟨*statement*⟩ } '`}`'

⟨*statement*⟩ ::= '`send`' ⟨*msg*⟩ '`to`' ⟨*dest*⟩
| '`reply`' ⟨*msg*⟩
| '`start`' ⟨*timer-id*⟩
| '`stop`' ⟨*timer-id*⟩
| ...

4.1.2 Semantics

As the asynchronous timing model has no notion of global time and local or global durations, our semantic model does not include a concept of time either. Therefore, actual durations of timers are not modelled and left out in the syntax. Under these conditions, the semantics of the timer functions are simple: as soon as a timer has been started, the corresponding timeout action is eligible for execution. As soon as a timer has been stopped, the corresponding timeout action is not eligible for execution anymore. Timeout actions compete (with equal priority) for execution with actions from input-action pairs.

4.1.2.1 Time in the asynchronous model

Note that the introduction of timers without an appropriate modelling of time can result in executions which would not be possible if the duration of timers were considered, even when accepting arbitrary message delivery times.

Consider Algorithm 4.1. When initialized, task A starts its timer and sends a message to B, causing it to start its timer as well. As soon as B's timer has elapsed, it answers A. Depending on whether A's timeout happens first or B's message about its timeout is received first, the variable `result` will after

termination contain `"AB"` or `"BA"` (if there are no failures). In our model, both orders are possible, regardless the actual time the timers are initialized with, as this time is not modelled. However, if actually the duration of B's timer is longer than that of A, an execution resulting in `"BA"` would not possible in a model where at least local timing is consistent among tasks, that is, assuming all actions but timers take zero time (including message delivery), the order of execution should exactly follow the relative relation of the timeout intervals.

As the suggested modelling of timers only results in more possible executions than assumed in reality, it can be seen as a conservative approximation of reality: properties holding for all executions in the model also hold for all executions in a more realistic model.

If desired, the model could be extended to establish a causal order which cares about the correct order of timeout events but the gain in precision, for example for the termination analysis, is probably negligible. In practice, this order is achieved by the equal progress of local time in each task (up to a certain tolerance depending on the clocks' precision).

Timer A: { A }

```
Variables
    result := ""
    timer timerA

input init() {
    start timerA
    send start() to B
}

timeout timerA {
    result += "A"
}

input B_stopped() {
    result += "B"
}
```

Timer B: { B }

```
Variables
    timer timerB

input start() {
    start timerB
}

timeout timerB {
    send B_stopped() to A
}
```

Algorithm 4.1: Demonstration of unrealistic executions in a model not considering time

4.1.2.2 Simulation

The semantics of the new syntax can be simulated by existing syntax and its semantics. As in the present model the only way of eventually executing an action is by receiving a message, a timeout event is simulated by a timeout message. Whenever a task starts a timer, it sends the corresponding timeout message (named after the timer) to itself. For each timer, there is a special input-action pair to receive this timeout message, with the action being the timeout action specified for that timer. To model stopping of timers, in addition there is a

Timer function	Timer syntax	Simulation syntax
Declare timer	*declaration of timer* timerA	*declaration of Boolean variable* timerA
Start timer	**start** timerA	**send** timerA() **to** self timerA := true
Stop timer	**stop** timerA	timerA := false

Timeout action	**timeout** timerA {
	<action code>
	}
		input timerA() **when** timerA {
		timerA := false
		<action code>
		}
		
		input timerA() **when not** timerA {}

Table 4.1: Simulation of timer syntax

Boolean variable for each timer keeping track of whether the timer is running. The variable is set to `true` by the `start` statement and to `false` by the `stop` statement. The input-action pairs receiving timeouts are thus equipped with a `when` clause, checking for this variable. To get rid of timeout messages of stopped timers, additional input-action pairs matching timeout messages with the corresponding variable being false and an empty action are added. Table 4.1 shows the described translation of the new timer syntax into existing syntax.

4.2 Evaluation

To evaluate the new concept, we discuss how timers can be used to deal with the absence of expected messages and therefore represent a means of achieving fault tolerance. We start by showing with a slightly more involved example how the language extension is used in practice.

4.2.1 A fault tolerant two-phase commit protocol

The two-phase commit protocol as described in Chapter 3 (Algorithm 3.1) does not tolerate any failures: the loss of a single message or the failure of a participant or the coordinator can cause the protocol to get stuck. Different mechanisms can be added to tolerate some of these failures (see for example [TV07]).

The idea is to use timeouts to solve situations where the protocol is blocked. These situations occur when the coordinator or participants wait for messages that are lost or, because the sender has failed, not sent. For simplicity, we only consider fail-stop failures of messages and tasks. Fail-stop-return failures can also be tolerated, but we would have to consider what it means for tasks to restart, which is a different topic.

Algorithm 4.2 is an adaptation of Algorithm 3.1 that makes use of the timer functionality introduced in this chapter to identify situations where a task is

blocked. Dealing with failures requires the careful consideration of many different situations which is why the algorithm is much more complex than the simple version described before.

A first change is that participants abort directly after being initialized when their own vote is "abort". This ensures that the `t.abort()` operation is executed even if the final decision (which is already known by tasks voting to abort anyway) is never reached. Furthermore, a variable `decision` keeps track of the current knowledge about the decision and is updated accordingly. As a small optimization, with the variable `voting` the coordinator keeps track of whether the voting phase is still running. This results in `abort()` messages only being sent out once after receiving a `vote(abort)`.

The first situation of a task being blocked in the protocol can occur when a participant waits for the coordinator's `vote_request()`. Note that before having been asked for its vote, a participant can always change its vote. Thus, if the `vote_request()` message is not received after a certain time, the participant can safely decide to abort. To implement this, we let participants start a timer `t_WaitVoteRequest` after being initialized. If it receives a `vote_request()` before the corresponding timeout, it stops the timer, but on a timeout it decides to abort. As the coordinator's `vote_request()` can still arrive after a timeout though, the vote has to be updated.

The next critical situation can happen on the coordinator's side, waiting for the participants' votes. In case no "abort" is received but still not n `commit()` messages arrive, the coordinator is blocked. To solve this situation, it can make a decision without waiting for all votes. The only reasonable choice is to abort, as some participants might not be able to commit. Assuming that the participants who have not sent their vote have failed, it is also the correct choice regarding the original rule for the decision. To implement this behaviour, the coordinator starts a timer `t_ReceiveVotes` when it begins to wait for votes.

Finally, it can happen that a participant is blocked when waiting for the coordinator's decision, either because the coordinator has failed or the `abort()` or `commit()` message is lost. In this case, the participant cannot simply decide on its own what to do, as other participants might already have carried out the decision which it does not know of. One solution to this problem is to ask other participants for the decision (which of course requires that participants can contact each other). In case the coordinator has sent out its decision to at least one participant or a participant has decided to abort on its own, there is a chance of getting the decision from such a participant. Therefore, after having replied to a `vote_request()` and thus waiting for a decision, a timer `t_WaitDecision` is started. However, if the decision is already known (because of the own vote being "abort"), there is no need to start the timer. A timeout causes a `decision_request()` message to be sent to all other participants. A participant receiving such a message replies with `abort()` or `commit()` in case it knows the decision and thereby causes the requesting participant to perform the corresponding operation. As because of the reception of multiple such replies and also in other situations the `t.abort()` and `t.commit()` operations can be executed multiple times, we still assume them to be idempotent.

Using timeouts, the two-phase commit protocol now tolerates a row of task and message failures. It shall be noted, however, that there are still situations where the protocol can get stuck. This is for example the case when the coordinator fails before it has sent out its decision to any participant (or the corresponding messages are lost) but at least one participant has already answered to a `vote_request()` with "commit" and thereby lost the chance to decide for abort on its own. By introducing an additional *precommit* phase, the *three-phase commit protocol* [Ske81] can deal with such coordinator failures. It can be implemented with timers in the same way as the two phases of the two-phase commit protocol have been implemented in Algorithm 4.2.

With the two-phase commit protocol we have exemplarily shown how timers can be used to express the detection of non-arriving messages, with the possible causes being task failures as well as message failures.

However, we also note that the requirement of declaring a timer and having to execute start and stop commands is a bit cumbersome, or even prone to error, when, in a certain phase of a protocol, one only wants to wait for a certain amount of time before proceeding otherwise in the protocol. In the next chapter, we will see a syntactic extension which introduces dedicated support for this design pattern.

4.2.2 Termination analysis

By simulating the new syntax with existing syntax, we have shown that the extended language can be modelled by the same semantic model as the basic language introduced in Chapter 3. Therefore, also the termination analysis still applies as before.

Fig. 4.1 shows the message type graph of the two-phase commit protocol with timeouts. Note that the message type graph is constructed after the timer functions have been translated into basic syntax using Table 4.1. This is equivalent to connecting an action with a `start` statement to the corresponding timeout action.

As all message flows depicted by the message type graph are actually possible, it is equal to the message flow graph. Thus we have seen another practical example where the message type graph is a good (in this case perfect) approximation and from acyclicity we can conclude that the two-phase commit protocol with timeouts always terminates.

As can be seen at this more complex example, when algorithms get larger it is not that easy anymore to construct the message type graph and check for cyclicity by hand. Here an automatic analysis constructing the graph and testing for cyclicity eventually finds its practical application.

4.2.3 Spontaneous actions

Note that timers provide a limited way of letting tasks execute actions without receiving a message (disregarding timeout messages from simulation). It might seem that timers would create cycles in the message flow graph because a task

Coordinator: { C }

```
Variables
    t // local part of transaction
    v // own vote (commit/abort)
    count := 0 // commit count
    voting := true // accepting
     votes
    timer t_ReceiveVotes

input init() {
    if v = abort then
        send abort() to P
        t.abort()
    else
        send vote_request() to P
        start t_ReceiveVotes
    end
}

input vote(abort) when voting
 {
    stop t_ReceiveVotes
    send abort() to P
    t.abort()
    voting := false
}

input vote(commit) {
    count++
    if count = n then
        stop t_ReceiveVotes
        send commit() to P
        t.commit()
    end
}

timeout t_ReceiveVotes {
    send abort() to P
    t.abort()
    voting := false
}
```

Participant: P = { $P_1, \ldots, P_n$ }

```
Variables
    t // local part of transaction
    v // own vote (commit/abort)
    decision := ⊥
    timer t_WaitVoteRequest
    timer t_WaitDecision

input init() {
    if v = abort then
        decision := abort
        t.abort()
    start t_WaitVoteRequest
}

input vote_request() {
    stop t_WaitVoteRequest
    reply vote(v)
    if decision = ⊥ then
        start t_WaitDecision
}

timeout t_WaitVoteRequest {
    v := abort
    decision := abort
    t.abort()
}

input abort() {
    stop t_WaitVoteRequest
    stop t_WaitDecision
    decision := abort
    t.abort()
}

input commit() {
    stop t_WaitDecision
    decision := commit
    t.commit()
}

timeout t_WaitDecision {
    send decision_request() to P
}

input decision_request() {
    if decision = abort then
        reply abort()
    if decision = commit then
        reply commit()
}
```

Algorithm 4.2: The two-phase commit protocol with timeouts

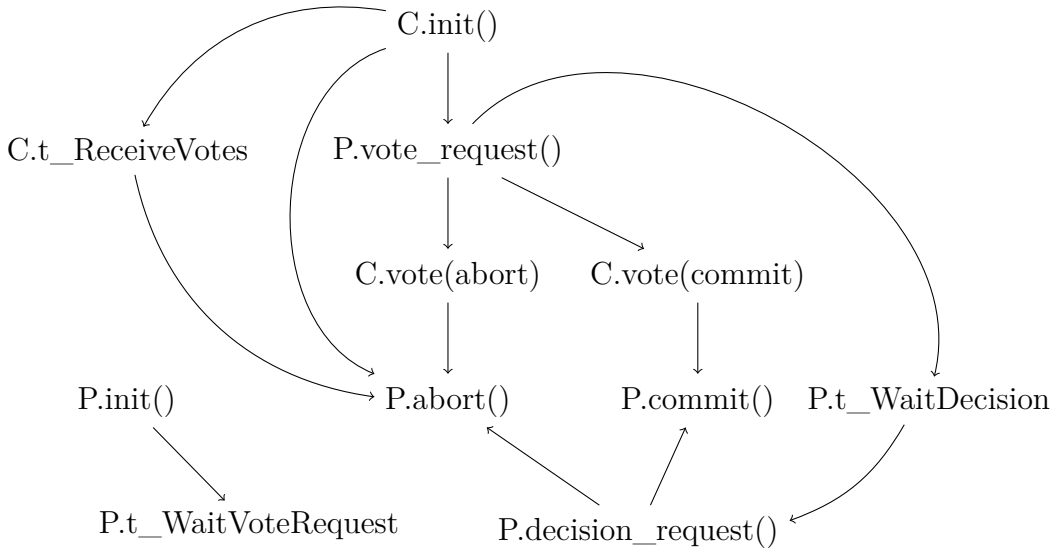


Figure 4.1: The message type graph of Algorithm 4.2 (timeout actions are named after the timers)

sends messages to itself. However, the message flow graph works on the level of actions and not on the level of tasks and thus there are only edges from actions with timers' `start` statements to the corresponding timeout actions. Only if timers are used to let a task perform an action in regular time intervals (by starting the timer again in its timeout action), a cycle would be introduced, like in the simulation of spontaneous actions (cf. Section 3.3.4.1). If, however, the number of repeated invocations can be limited, the idea of limited edges from Section 3.3.7.3 can be applied, making the cycle irrelevant for non-termination.

4.2.4 Real systems

In real implementations, timeout events should of course not be modelled as actual messages being sent. Messages are not necessary, as timers are always local. Therefore, timeout events are simply added locally to the input buffer. Timers are implemented by local clocks which are started and stopped by the corresponding commands and a timeout event is only created when the timer actually expires.

4.3 Conclusion

In this chapter, we have made the input-action language more expressive regarding an aspect of fault tolerance. We introduced timers as a way of limiting the waiting for messages to a certain amount of time. This allows for the (approximated) detection of lost or unsent messages and can thus also be used to detect task failures. With timers we were able to make the two-phase commit protocol

tolerant against various failures and could still prove termination using the message type graph.

We expect that the dedicated syntax for the recognition of failures with timeouts is a better starting point for static analysis on fault tolerance. One reason is the specific timeout action which will be executed when a message is lost: an analysis can assume the case of a message being lost and base further considerations on the fact that a certain action will be executed.

Certainly one can think of further language extensions to improve analyzability, for other aspects of fault tolerance as well as for completely different non-functional properties. With the extension in this chapter we have shown that the input-action language is easily extensible and that our model is general enough to appropriately express a non-trivial extension like timers (we accept a small imprecision regarding possible orders of timeout events), even by simulation with existing syntax.

CHAPTER 5

Usability

In this chapter, we want to take a look at how convenient it is to formulate distributed algorithms in the input-action language including its extension with timers. We identify some problems regarding the scalability to larger programs and introduce a new syntactic element to deal with it: *protocol stages* allow to describe a protocol's course of events to whatever degree is appropriate. The resulting potential of specifying structural properties does not only provide a better overview but has a row of additional advantages.

5.1 Deficiencies of the input-action language

We have argued, and shown with some examples, that distributed algorithms in general can be formulated relatively easy in the input-action language. The main reason is that it can directly reflect the reactive behaviour of algorithms using input-action pairs. Guards allow to delay the processing of a message until a certain condition holds. This way the language abstracts from the order in which messages arrive, for which in the asynchronous timing model there is no guarantee anyway.

However, in its current form, the language also has some drawbacks. When an algorithm gets more sophisticated or a protocol includes more steps, the number of input-action pairs per task will rise. As there is no way of structuring input-action pairs syntactically, one can quickly lose the overview. Moreover, the extension with timers adds another component which makes it difficult to picture the possible control flow, as can be seen at the two-phase commit protocol with timeouts (Algorithm 4.2): the execution of a `start` command anywhere can result in the activation of another action at some later point but only provided the `stop` command has not been executed. While this kind of control flow is inherent to the input-action language, some "scoping" is possible and better ways exist to express that in a certain phase of a protocol the waiting for a message should be limited in time.

In the following, we suggest a syntactic extension which allows to reflect the sequential course of a distributed protocol. Informally, we understand a *distributed protocol* as an algorithm which can be divided into sequential phases, as for example the two-phase commit protocol. In general, the phases do not have to be traversed linearly though, one can imagine an arbitrary graph of transitions between those phases. By accounting for the design patterns of this class of algorithms, the above problems of structuring and specification of timeouts are

addressed, resulting in a number of advantages regarding usability as well as analyzability. We start by presenting the new syntax, show how it can be used in accordance with the semantic model from Chapter 3 and finally evaluate the usefulness of the new concept.

5.2 Protocol stages

Distributed protocols can naturally be divided into *stages*, each of which represents a certain phase of the protocol, with its specific functionality and goals. We want to introduce the syntactic possibility of integrating stages into tasks. This is done by extending a task's state with its current *protocol stage*. Transitions to other protocol stages are performed via explicit transition statements in actions. A suitable division into stages for the protocol at hand has to be chosen by the designer. The additional structure is eventually gained by associating input-action pairs with stages. An input-action pair is then only active when the task is in the corresponding stage.

5.2.1 Syntax

Before explaining the semantics of stages in detail, we describe the new syntactic elements. First of all, input-action pairs must now be enclosed by a named *stage* block which in addition allows for the definition of a *stage initializing action* as well as a *stage timeout action*. Statements now also include a *stage transition statement* ($\rightarrow$). In addition, for better readability, we now allow to omit the enclosing parentheses of actions consisting of only one statement. Like for task identifiers, we leave open the concrete form of stage identifiers.

The following EBNF grammar describes the extended input-action language, based on the syntax from Section 3.2.4. It does not include explicit timers as introduced in Chapter 4 anymore, that is, there are no timer declarations, start and stop commands. The replacement of their functionality by stage timeouts is discussed in Section 5.3.3.

⟨*task*⟩ ::= '`task`' ⟨*task-identifier*⟩ '`{`'
⟨*variable-declarations*⟩
{ ⟨*stage*⟩ }
'`}`'

⟨*stage*⟩ ::= '`stage`' ⟨*stage-identifier*⟩ '`{`' { ⟨*task-element*⟩ } '`}`'

⟨*task-element*⟩ ::= ⟨*input-action-pair*⟩
| ⟨*init-action*⟩
| ⟨*timeout-action*⟩

⟨*input-action-pair*⟩ ::= ⟨*input*⟩ ⟨*action*⟩

⟨*init-action*⟩ ::= '`init`' ⟨*action*⟩

⟨*timeout-action*⟩ ::= '`timeout`' ⟨*action*⟩

⟨*input*⟩ ::= '`input`' ⟨*msg-matcher*⟩ ['`when`' ⟨*boolean-exp*⟩]

⟨*action*⟩ ::= '`:`' ⟨*statement*⟩
| '`{`' { ⟨*statement*⟩ } '`}`'

⟨*statement*⟩ ::= '`send`' ⟨*msg*⟩ '`to`' ⟨*dest*⟩
| '`reply`' ⟨*msg*⟩
| '→' ⟨*stage-identifier*⟩
| ...

5.2.2 Semantics

We now describe the exact semantics of stages before showing how it can be simulated by existing syntax and semantics.

At all times, a task is in exactly one of the specified stages. The first stage specified becomes the initial stage. The current stage is not directly accessible by the task's code. The only way to modify it is using the transition statement. Apart from changing the current stage, a transition also results in the execution of the target stage's *initializing action*.

The effect of the current stage on a task is that it limits the currently applicable input-action pairs. More precisely, when a task is in a certain stage, its behaviour is equivalent to a task where only the input-action pairs defined in the section of that stage are present. Switching stages thus enables and disables input-action pairs.

There are no explicit timers anymore, but stages have *timeout actions* instead. The transition to a stage now implicitly starts a timer for this specific stage. Its timeout is received by the stage's optional timeout action. As soon as the stage is left, the timer is stopped (and started anew when coming back to the stage).

Note that messages received which cannot be handled by the current stage's input-action pairs stay in the input buffer and might be handled when switching to another stage. How this kind of *scoping* can be useful is discussed in Section 5.3.4

Simulation

The semantics of stages can be simulated using the basic syntax for the input-action language from Chapter 3 plus the timer extension from Chapter 4.

The current stage of a task is represented by a variable and each input-action pair is equipped with an additional condition in its guard, namely whether the current stage equals the one the input-action pair is associated with.

Stage timeouts are simulated by timers. For each stage, a timer with the stage's name is declared. The transition to a stage starts this timer. The timeout action of a stage is represented by a timeout action for the corresponding timer. Transiting to another stage stops the timer of the previous stage, updates the stage variable, executes the code specified in the target stage's initializing action and starts the new timer. If a stage has no timeout action, there is no need to declare a timer for it. Table 5.1 shows the described translation of the new syntax into existing syntax.

Stage functionality	**Stage syntax**	**Simulation syntax**
Current stage	-	*declaration of variable* stage

Stage definition	**stage** StageA {
	...
	input m() **when** <cond> {
	<action code>
	}
	...
	}
		...
		input m() **when** <cond>
		&& stage = StageA {
		<action code>
		}
		...
		declaration of timer
		StageA

Initializing action	**stage** StageA {
	...
	init {
	<StageA **init** action>
	}
	...
	}
		see stage transition

Stage timeout	**stage** StageA {
	...
	timeout {
	<action code>
	}
	}
		timeout StageA {
		<action code>
		}

Stage transition	**stage** StageA {
	...
	... {
	<action code>
	−> StageB
	}
	...
	}
		... {
		<action code>
		stop StageA
		stage := StageB
		<StageB **init** action>
		start StageB
		}

Table 5.1: Simulation of stages

5.3 Evaluation

We evaluate the concept of integrating stages into tasks with the help of the two-phase commit protocol, demonstrating all new features introduced in this chapter.

5.3.1 The two-phase commit protocol with stages

In the previous chapter, we have implemented a fault tolerant version of the two-phase commit protocol using timers. We already noted that the manual specification of timers requires a lot of code that might not be necessary for the recurring design pattern of limiting the waiting for messages in a certain phase of a protocol. Furthermore, the concurrent use of multiple timers can quickly get confusing and lead to mistakes. By reformulating Algorithm 4.2 with the use of stages (Algorithm 5.1), we show that stages can indeed save a lot of redundant code and facilitate the formulation of such more complex protocols in general.

To start with, note how the additional structure helps in reading the algorithm: one can first identify the stages and their possible transitions (Fig. 5.1) and then figure out what the input-action pairs of each stage do.

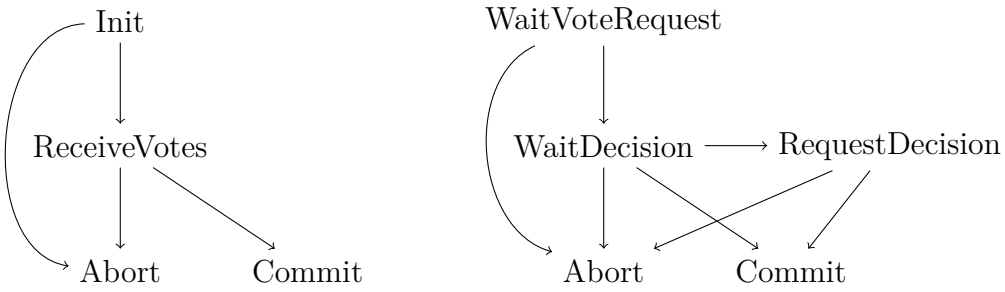


Figure 5.1: Stages with possible transitions of the two-phase commit protocol (Algorithm 5.1) for coordinator (left) and participants (right)

We now describe the two-phase commit protocol in terms of stages, meanwhile explaining the new features and how they are useful for the formulation of the protocol.

The coordinator starts in stage `Init` where, in case it is self ready to commit, it sends the `vote_request()` message to all participants and transits to stage `ReceiveVotes` where it waits for the participants' votes. Otherwise it directly transits to stage `Abort`. To deal with failures of participants while waiting for votes, it makes use of a stage timeout resulting in a transition to stage `Abort`. If the coordinator receives an `abort()` message, it also transits to stage `Abort`. When it has on the other hand received all n `commit()` messages before the timeout, it transits to stage `Commit`. The stages `Abort` and `Commit` represent the possible outcomes (and final states) of the protocol (at both, the coordinator and the participant task) and, with the stage initializing actions, are used to perform the corresponding actions: the coordinator sends out the `abort()` or `commit()`

messages and participants and coordinator perform the respective operations on their local part of the transaction.

The Participant task starts in stage `WaitVoteRequest`. In contrast to the timer version which can receive delayed `vote_request()` messages even after a timeout, here we only want to react to this request in its dedicated stage. After a timeout and the resulting transition to stage `Abort`, a `vote_request()` will not be answered anymore. Assuming that a timeout corresponds to a coordinator failure, this decision is reasonable. In case the coordinator has not failed, it will reach the decision to abort also without this participant's vote, either by other "abort" votes or by a timeout because of the missing vote. This design choice, encouraged by the nature of stages, has the advantage of not having to consider many special cases, like caring about updating the own vote, which could easily be overlooked. Stages thus provide a kind of *scoping* which can help in avoiding design mistakes.

When a `vote_request()` is received, the participant sends its vote and directly transits to stage `WaitDecision` in case its vote is "commit" or to stage `Abort` otherwise. In stage `WaitDecision`, it waits for the coordinator's `abort()` or `commit()` message and on the reception of either transits to the corresponding final stage. If no decision is received, a timeout causes a transition to stage `RequestDecision` where `decision_request()` messages are sent out to other participants to find out about a potential decision. Their `abort()` or `commit()` replies also result in a transition to the corresponding final stage.

The final stages in the Participant task have an additional purpose: they do not only represent the outcome of the protocol but also communicate the decision to other participants, namely those being in stage `RequestDecision` and asking for it. Here we have an example of how the same message can have a different effect when received in different stages: `decision_request()` is answered by `abort()` or `commit()`, depending on whether the receiving Participant is in stage `Abort` or `Commit`. If none of these final stages has been reached yet, the request is even automagically stored and answered as soon as one of the stages is entered, whereas in Algorithm 4.2 it would be ignored. The usage of stages to answer a request properly makes the `decision` variable used in Algorithm 4.2 for that purpose unnecessary, as its values are already encoded in the current stage: its value would be `abort` in stage `Abort`, `commit` in stage `Commit` and $\perp$ in other stages.

Along the transformation, we have automatically gained some further advantages. Most importantly, the non-trivial (and with manual starting and stopping also verbose) managing of timers in Algorithm 4.2 has become obsolete, by tightly coupling timeouts with stages. Furthermore, the `voting` variable is not necessary anymore for the coordinator to keep track of whether the protocol is still in the voting phase – this is already represented by the `ReceiveVotes` stage. In addition, the `t.abort()` and `t.commit()` operations do not have to be idempotent anymore, as they can only be executed once when entering the `Abort` or `Commit` stage.

Coordinator: { C }

```
Variables
    t // local part of transaction
    v // own vote (commit/abort)
    count := 0 // commit count

stage Init {
    init {
        if v = commit then
            send vote_request() to P
            → ReceiveVotes
        else
            → Abort
        end
    }
}

stage ReceiveVotes {
    input vote(abort):
        → Abort

    input vote(commit) {
        count++
        if count = n then
            → Commit
    }

    timeout:
        → Abort
}

stage Abort {
    init {
        send abort() to P
        t.abort()
    }
}

stage Commit {
    init {
        send commit() to P
        t.commit()
    }
}
```

Participant: P = { $P_1, \ldots, P_n$ }

```
Variables
    t // local part of transaction
    v // own vote (commit/abort)

stage WaitVoteRequest {
    input vote_request() {
        reply vote(v)
        if v = commit then
            → WaitDecision
        else
            → Abort
    }

    timeout:
        → Abort
}

stage WaitDecision {
    input abort():
        → Abort

    input commit():
        → Commit

    timeout:
        → RequestDecision
}

stage RequestDecision {
    init:
        send decision_request() to P

    input abort():
        → Abort

    input commit():
        → Commit
}

stage Abort {
    init:
        t.abort()

    input decision_request():
        reply abort()
}

stage Commit {
    init:
        t.commit()

    input decision_request():
        reply commit()
}
```

Algorithm 5.1: The two-phase commit protocol with timeouts making use of stages

5.3.2 Structuring tasks using stages

On first sight, stages seem to make the formulation of an algorithm considerably larger by adding boilerplate code. However, it shall be noted that protocols can be divided into stages with different granularity. For example, in Algorithm 5.1 the stages `Abort` and `Commit` can be omitted in the Coordinator task – the code of the initializing actions would then simply be put where for now the transitions to these stages are made. The advantage of adding the stages anyway is the additional structure, which for example allows to get an idea of the protocol by just looking at the stages and the transition graph (Fig. 5.1).

If a distributed protocol is syntactically structured using stages, the process of understanding the algorithm can be divided into a top-down approach consisting of two steps: first the basic procedure can be read off the stage transition graph and then each stage can be examined in detail. This is especially useful when algorithms get even larger. The same argument holds for the design process: an algorithm can be designed with a top-down approach by first finding appropriate stages and then implementing the stages with input-action pairs. It is the task of the designer to find a compromise between number of stages and complexity of each stage.

Stages also provide a limited way of abstracting code in a way similar to procedures in procedural languages: the code of a stage's initializing action can be executed by transiting to the stage with a single statement, which can be seen as a "call". In the two-phase commit protocol, this possibility is used with the `Abort` stage, which can be transited to from three and four different places in Coordinator and Participant task, respectively.

Indeed, performing actions only on entering a new protocol stage seems to be a common pattern. Reducing all actions of an algorithm to stage transitions and stage initializing actions could be a restriction that allows for better analyzability. The result can be seen as a (hierarchical) state machine where each state has substates representing the task's local variables. Transitions are triggered by incoming messages or, after state changes, by messages in the input buffers. Transitions can result in local state changes (that is, transitions of the substate) and the sending of messages to other state machines.

Finally, with (among others) the variable `decision` in Algorithm 4.2 becoming redundant when using stages to formulate the two-phase commit protocol and the opportunity of more intuitively formulating the handling of the `decision_request()` message coming along with it really shows the power of stages to express the semantic properties of a protocol.

The described syntax for stages also has a disadvantage: if an input-action pair is supposed to be active in multiple stages, it would have to be copied into their definitions, resulting in code duplication, like with the `abort()` and `commit()` input-action pairs in stages `WaitDecision` and `RequestDecision` in Algorithm 5.1. However, the explicit mentioning of which messages are expected in which stage can also be seen as an advantage, as it results in a more precise description of the protocol. A way around the code duplication is an alternative syntax which allows to associate multiple stages with an input-action pair. This can be achieved by tagging input-action pairs with stages or by allowing to

reference input-action pairs from other stages.

5.3.3 Replacing timers with stage timeouts

On first sight, leaving out explicit timers seems like a restriction. Keeping the former timer syntax (which technically would be no problem) is not necessary though as in a protocol, a timeout can usually be associated with a stage. If more fine-grained timing within a stage is required, the stage can be split into multiple stages. It is, however, not possible to let a single timer run over multiple stages. This can be useful if a protocol consists of a series of stages where independent of in which stage a task currently is, an overall timeout is desired. If considered important, a small syntactic addition can solve this problem: we can allow stages to specify which stages' timers should continue when entering it. On a transition from stage A to stage B where B specifies to continue A's timer, B's timer is started with the remainder of A's timer and on a timeout, A's timeout action is executed (or, alternatively, a replacement specified in B).

Another enhancement allowing for combined timers would be *hierarchical stages*, where a set of stages can be grouped to have an overall timeout. The overall timer is then not affected by transitions of sub-stages.

Using the new form of declaring timeouts can be seen as more natural and compact and thus easier to write and read: when timeouts are directly associated with the phases of a protocol, the code for declaring, starting and stopping timers would always be similar while at the same time the designer has to be careful to not forget anything. The division of a protocol into stages reduces the complexity of managing timers, which could otherwise run in parallel, whether intended or not.

5.3.4 Managing unexpected messages

In correspondence to the procedure of a protocol, stages also provide a structured way of specifying which messages are expected during which phase of a protocol and what should be done with unexpected messages.

A first possibility resulting from scoping is the synchronization of a protocol: a coordinator might allow participants to go on in the protocol even if the current phase is not completed because the coordinator is still waiting for answers of other participants. Then the participants having progressed further can already send messages to be processed in a later stage of the coordinator. The management of the current stage in the coordinator's task implicitly organizes the holding back of these messages until it has transited to the respective stage.

Furthermore, scoping can be useful when in a stage certain messages should not be received by design and the actual reception would reveal an implementation error. Or a task wants to protect itself from other tasks not behaving according to the protocol or simply ignore messages which are no more relevant in the current stage (like for example a second `abort()` message received by the coordinator in the two-phase commit protocol).

These situations can be dealt with in different ways. In the case of implementation errors, a runtime exception (or some other error handling procedure) might be appropriate. A runtime exception can be simulated by a transition into an error stage with no input-action pairs what conceptually is equivalent to the task having halted or failed (recall that other tasks cannot notice whether messages actually arrive). This behaviour can be implemented by adding input-action pairs matching all unexpected messages with an action transiting to an error stage. To ignore a certain type of message, it suffices to add an input-action pair matching it but with an empty action.

As the dealing with unexpected messages can be seen as a design pattern, one could also introduce specific syntax stating for example that all message types not occurring in any input part of the stage's input-action pairs should result in a transition to an error stage. Also a more fine-grained specification of messages not allowed (or of those allowed only) in a certain stage is imaginable, for example as a list of message types or message matchers. The same can be done for messages to be ignored. The simulation of this new syntax via input-action pairs is trivial (only the according input-action pairs as described above have to be generated).

The explicit specification of unexpected messages also opens up new possibilities for static analyses: potential exceptions because of unexpected messages or messages always ignored might be detectable statically in certain cases.

5.3.5 Stages and analyzability

Finally, we want to discuss the consequences of stages regarding analyzability.

First of all, as the new syntax can be simulated by the syntax from Chapter 4 (which again is simulated by the basic syntax from Chapter 3), the termination analysis from Chapter 3 still applies. As the message type graph of the stage version of the two-phase commit protocol is still acyclic (Fig. 5.2), we can conclude that also this version always terminates. However, we can also see that the message type graph is not powerful enough to profit from stages: the input-action pairs are considered without their relation to stages. This is clear considering that the simulation simply consists of adding an additional condition to the input-action pairs' guards, which have no influence on the message type graph. More sophisticated analyses would also take the syntactic form of stages into account to benefit from the additional structure.

Regarding analyzability in general, it is (intuitively speaking) the lack of structure which limits opportunities for static analyses on distributed protocols in the original input-action language: input-action pairs are specified with little relation to each other, though often semantic relations exist. Stages are a way of expressing the structure and dynamic behaviour of a protocol. Also the more abstract way of specifying timeouts might be an advantage for static analyses. We expect this to be useful for several static analyses on non-functional properties, including termination.

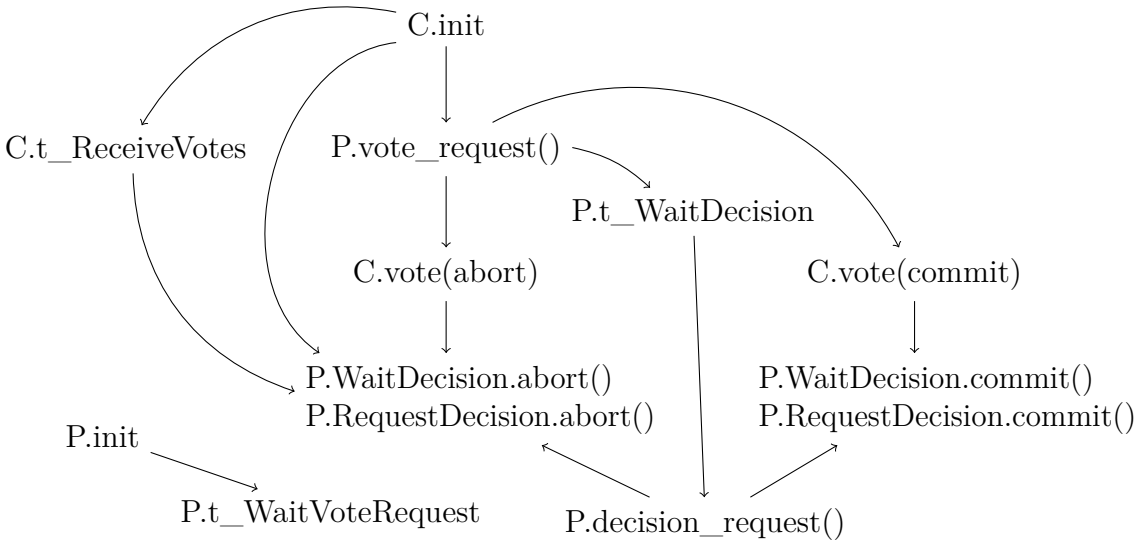


Figure 5.2: The message type graph of Algorithm 5.1 (C.init and P.init are the (possibly empty) initializing actions of the tasks, the abort()/commit() actions are shown as one node as they have the same incoming and outgoing edges, timeout actions are labelled according to the stage)

5.4 Conclusion

By integrating stages into tasks, we have introduced a useful abstraction that accounts for several design patterns occurring in distributed protocols.

First of all, we have now the syntactic means of structuring a task according to a protocol's course of events by associating input-action pairs with stages. A better overview (not least through the visualization with the stage transition graph), making the language more practicable when complexity increases, is only one advantage. Stage timeouts allow to conveniently implement the common design pattern of limiting the waiting for messages during a certain phase of a protocol. This results in improved readability and can reduce the chance of error during design. Finally, stages are a promising concept for scoping expected messages: they allow to specify which messages are expected at a certain phase of a protocol and open up new possibilities of dealing with unexpected messages. All in all, stages allow for a more structured control of a task's functionality and thus ease development.

Potential for further development of the concept exists. Hierarchical stages as a generalization of grouping into multiple hierarchies can again address structural design patterns. Another concept, which can be combined with stages, is *task inheritance* which, similar to classes in object-oriented programming, allows tasks to inherit functionality from other tasks.

We also expect algorithms formulated using stages to be more accessible to static analyses in general, especially because of putting dynamic behaviour into a syntactic frame and the more uniform specification of timeouts. This might be especially interesting for a termination analysis when in addition to termination

also *progress* is considered, that is, a protocol has to reach a certain final state. Furthermore, the scoping introduced with stages can allow for additional analyses on potential implementation errors regarding which messages are expected when.

Whether an algorithm can profit from stages depends on its nature. The better a protocol can be divided into different phases, the more it will benefit from using the syntactic elements introduced in this chapter.

CHAPTER 6

Implementation

In this chapter, we introduce JIAL (Java input-action language), a programming language based on the abstract specification language described in Chapter 3. To this end, we first specify a concrete syntax. We then describe a compiler together with a runtime system, building a framework to execute and simulate distributed algorithms written in JIAL. Finally, we implement the termination analysis based on message type graphs.

6.1 Towards a programming language

To actually implement and execute algorithms, the abstract specification language described so far is not sufficient, as it only describes the main syntactic elements characterizing the input-action language. However, as already noted, it can easily be extended with additional statements to obtain a full programming language.

Our approach is to integrate an existing general-purpose imperative language (from now on referred to as the *base language*) into the syntax frame from Section 3.3.6.1 (the basic input-action language with the use of message types). To this end, in the ⟨*variable-declarations*⟩ part of a task we allow all declarations of the base language and for additional statements (⟨*<statement>*⟩) all its statements. Message types are implemented using type signatures: a name with a list of parameter types.

As our base language we choose Java, a highly portable, commonly used general-purpose programming language. It also serves as the implementation language for our runtime system and as intermediate language for our compiler, before obtaining Java bytecode as an executable result. We emphasize that the goal of this implementation is not efficiency or suitability for specific practical use cases but rather a demonstration of the concept's working. We call the resulting language JIAL, which is short for *Java input-action language*.

To focus on the main concepts, here we only consider the basic input-action language as described in Chapter 3. The language extensions from chapters 4 and 5 can be incorporated easily, with the syntax presented in those chapters.

6.2 Syntax

An implementation of a distributed algorithm in JIAL consists of two parts: a message type declaration and task declarations. Both require the specification of a

package name (⟨*package-name*⟩) and allow to import classes from other packages (⟨*imports*⟩), with the syntax being exactly that of Java class file headers.

6.2.1 Message type definitions

Message types are defined in a dedicated file with the following syntax, where '`\n`' denotes the newline character. A message type consists of a name and a list of parameter types. Names are only assigned to parameters when using the message type in input-action pairs' guards. ⟨*msg-type-name*⟩ is a normal Java identifier that can be used for classes and uniquely identifies the message type. ⟨*java-type*⟩ must be a primitive data type or the name of a class in scope.

⟨*msg-type-file*⟩ ::= ⟨*package-name*⟩
⟨*imports*⟩
'`{`' ⟨*msg-type-def*⟩ '`\n`' '`}`'

⟨*msg-type-def*⟩ ::= ⟨*msg-type-name*⟩ '`(`' [⟨*java-type*⟩ { '`,`' ⟨*java-type*⟩ }] '`)`'

6.2.2 Task definition

The following EBNF describes the syntax for a task file in Jial, with some elements explained below. Note that for the sake of readability, we omit details like possibly required spaces.

⟨*task*⟩ ::= ⟨*package-name*⟩
⟨*imports*⟩
'`task`' ⟨*task-identifier*⟩ '`{`'
⟨*java-declarations*⟩
{ ⟨*input-action-pair*⟩ }
'`}`'

⟨*input-action-pair*⟩ ::= ⟨*input*⟩ ⟨*action*⟩

⟨*input*⟩ ::= '`input`' ⟨*msg-type*⟩ ⟨*var-list*⟩ ['`when`' ⟨*boolean-exp*⟩]

⟨*action*⟩ ::= '`{`' { ⟨*statement*⟩ } '`}`'

⟨*var-list*⟩ ::= '`(`' ⟨*java-type*⟩ ⟨*java-var*⟩ { '`,`' ⟨*java-type*⟩ ⟨*java-var*⟩ }
'`)`'

⟨*statement*⟩ ::= '`send`' ⟨*msg*⟩ '`to`' ⟨*dest*⟩ '`;`'
| '`reply`' ⟨*msg*⟩ '`;`'
| ⟨*java-statement*⟩

⟨*msg*⟩ ::= ⟨*msg-type*⟩ ⟨*value-list*⟩

⟨*dest*⟩ ::= ⟨*integer-set-exp*⟩
| '`$`'⟨*task-identifier*⟩
| '`$ALL`'

⟨*task-identifier*⟩ must be a valid Java identifier for classes. ⟨*java-declarations*⟩ covers all Java declarations including variable declaration with assignment as well as class definitions. ⟨*msg-type*⟩ must be the name of a message type declared in the message type file. A ⟨*var-list*⟩ is a normal Java parameter list as used in method signatures, representing message signatures. Similarly, a ⟨*value-list*⟩ is then a comma-separated, parenthesed list of Java expressions as used in method calls, the types of which have to match the type signature of the specified message type. A ⟨*boolean-exp*⟩ is a Java expression of type `boolean`, a ⟨*java-statement*⟩ is any Java statement. The Java code in both, ⟨*boolean-exp*⟩ and ⟨*action*⟩, may access the corresponding message's parameters and its metadata (see "Special variables") as well as all declarations from the ⟨*java-declarations*⟩ section. Our implementation only supports direct addressing via ID. For this purpose, ⟨*integer-set-exp*⟩ is a Java expression of type `Set<Integer>`, to represent a set of task IDs to be the destinations of a message. Depending on the communication module in use, IDs can, however, also represent pseudonyms. To allow for full anonymous systems, an alternative form of addressing has to be implemented and supported by the communication module but otherwise no changes of the implementation are necessary.

Specifying a task with this syntax defines a *task type*. An algorithm can then make use of multiple *instance*s of this type.

Special variables

Depending on the context, special variables are available. First of all, `$ID` always refers to the task's own ID. Regarding addressing, it is useful to be able to send a message to all tasks of the same type or even to all tasks of the algorithm. To this end, for each defined task `T`, there is a variable `$T` of type `Set<Integer>` including the IDs of all instances of this task. Similarly, the special variable `$ALL` refers to all tasks.

Inside guards and actions, special local variables are available referring to the currently processed message's parameters and its metadata. Message parameters are accessed via the names provided in the ⟨*var-list*⟩ part of an input-action pair (the definition of input-action pairs can thus be read like method definitions) and `$src` is the ID of the current message's sender.

Initialization

For initialization, each task can use the special input-action pair `input init()`, with any custom action. An `init()` message is sent to all tasks when the algorithm is initialized. For an algorithm to start sending messages, it has to make use of this initializing action in at least one task. Regular guards can be used, for example to only let a certain instance of a task take an initial action.

6.3 Compiler

We implemented a compiler in Haskell, a functional language well-suited for the implementation of compilers. The functional paradigm especially lends itself to the transformation of data structures into other data structures, like it is done for trees in compilers. Haskell itself comes with a huge collection of packages for source manipulation, for specific languages as well as a variety of general tools like lexer and parser generators required to build a compiler toolchain.

6.3.1 Architecture

The overall process of translating Jial to executable Java bytecode incorporates two steps. The first consists of compiling Jial to plain Java which is the task of the compiler described in this chapter. The second step consists of compiling the resulting Java source code to Java bytecode which is done by the Java compiler. As the base language of Jial is Java, only the syntactic parts specific to our language (as given in the above syntax definition) have to be specifically processed by our compiler whereas blocks of Java code are simply identified as such and then kept as-is. To glue Java code generated for our specific syntax together with the native Java code blocks, we use a small runtime system written in Java, providing basic classes and functionality.

6.3.2 Lexing and parsing

The lexer and parser generator tools we use are *Alex*[1] and *Happy*[2], some standard tools for Haskell. Alex is similar to lex or flex for C/C++, tokenizing based on regular expressions, and Happy is similar to yacc for C, producing a parser based on a BNF grammar.

The lexer's task is to detect comments (line and block comments), strings, keywords and specific syntactic elements of the input-action language as well as blocks of Java code and produce the according tokens. To this end, we use a stateful lexer which remembers a current lexing mode while scanning the input stream. We also require the lexer to retain most whitespace such that the later intermediate representation in Java is still formatted like the original source and can thus be used for further inspection.

The parser takes the token stream generated by the lexer and produces an abstract syntax tree according to a BNF grammar which is based on the EBNF grammar from Section 6.2.

6.3.3 Runtime system

The runtime system provides the basic functionality required by tasks (in the form of a base class `Task`), like storing received messages, executing actions as soon as a message matching a guard is present and sending messages.

[1] https://www.haskell.org/alex/
[2] https://www.haskell.org/happy/

The runtime system makes use of a communication module which manages the sending and receiving of messages between tasks. It is responsible for serialization and network communication (or its simulation) as well as for assigning IDs to tasks and then allow them to communicate via these IDs. The actual implementation of the communication module is not part of the runtime system – requirements are too specific to provide an all-in-one solution. Instead, the user can provide their own communication module by implementing the above functionality.

Note that while in our model of distributed algorithms we assumed input buffers to have infinite capacity, this cannot be realized in practice. Instead, the communication module has to decide how to deal with situations where the maximum capacity of a buffer is reached. A simple solution is to drop messages which cannot be stored by the destination anymore. Of course this results in a deviation in semantics regarding the faultless model but as one would usually not assume the communication module to be faultless anyway, it is a reasonable choice. Another possibility is to block the sending task until the receiving task can store the message. However, this requires appropriate synchronizing mechanisms and can introduce additional deadlocks when a circular blocking situation occurs.

6.3.4 Code generation

Given the abstract syntax tree produced by the parser, it is straight-forward to generate the eventual Java code. This is demonstrated with an example in Section 6.8.

To start with, the file with the message type definitions is translated to a Java class `M` containing for each message type a class with the given parameters as fields. These message type classes derive from a common class `Message` which contains fields for messages' metadata.

Following, for each task a Java class with the given name ($\langle$*task-identifier*$\rangle$) is generated, deriving from the base class `Task` provided by the runtime system. First of all, from a task file the Java code blocks outside of input-action pairs, including the header of the file with package name and imports, are placed as-is into the new class file. Then, for each input-action pair, two methods are generated representing guard and action. Both methods are provided with the message in question as parameter. At the beginning of each method, variables for the message's metadata as well as for the message parameters are established, with the names given in the input part's parameter list. The guard method simply consists of the Boolean expression from the input-action pair's `when` part. It thus returns `true` iff the corresponding input-action pair is eligible for execution with the given message. The action method simply contains the code provided in the source's corresponding action part where only `send` and `reply` statements have to be handled specifically. The syntax of both statements requires the specification of a message type and the corresponding parameters. The form of this specification is identical to a constructor call in Java, except for the missing `new` keyword. Thus, by inserting the `new` keyword just before the specification of a message to be sent, we obtain a valid `Message` object. Furthermore, some code to set the message's metadata has to be generated. The sender is set to the

current task's ID, the receiver is set according to the expression in the `to` part (in case of the `send` statement) or (in case of the `reply` statement) to `$src`, the ID of the current message's sender. Finally, the new `Message` object is passed to the communication module to be delivered to the respective destination(s).

In addition, a `prepare()` method is generated to initialize a task's special variables (like the ID sets of all task types) and register the generated guard and action methods with the base class. It can be called after the task has been registered with the communication module.

The calling of the input and action methods is coordinated by a `step()` method which lets the task take a step (that is, execute an action with a satisfied guard) if possible. A mapping from message types to pairs of guard and action methods is used to determine which actions are currently applicable for a certain message. There is no given order in which input-action pairs are checked.

The result of the code generation is the message type class and one class per task representing that task's functionality. Together with the runtime system package, these can be directly used in a Java project. Depending on the desired behaviour, the runtime system can call a task's `step()` method whenever a message has been received and repeat the call until no more steps are taken.

6.3.5 Error reporting

There are two independent levels of error reporting: syntax errors according to the specific syntax of Jial and syntax errors in blocks of Java code. The latter are reported by the subsequent Java compiler, the former by our own tools. There are also cases of syntax errors in between pure Java syntax errors and the violation of specific Jial syntax: for example, the declaration of a "reserved" variable name can conflict with a variable declared by code generation. Syntax errors of such categories are eventually reported by the Java compiler as well.

We did not implement sophisticated error reporting. While the lexer reports the position in a source file which causes a lexing error, the parser outputs the tokens it was not able to parse, without giving concrete reasons or hints to solve the problem. As the syntax of Jial (excluding the Java part) is quite simple and thus syntax errors should be easily detectable by sight, we think this is not a major limitation when developing in the language.

To facilitate debugging, the compiler maintains most of the formatting of the original source. Thus, when the Java compiler references a syntax error, the corresponding location in the original source file can be found easily.

6.4 Simulation

We implemented a simulator which allows to execute the entire algorithm in one program, on a single machine. To this end, it provides a communication module which simply transfers messages between the local `Task` objects. In addition, it keeps a record of all events, that is, messages sent and actions executed. The simulator can therefore be used as a basis for dynamic analyses on non-functional properties. It can also be used to test algorithms for correctness by providing

certain inputs and checking the task states after termination. For improved performance, the `step()` methods of the different tasks can be run in different threads.

As a more advanced feature one can think of a "simulation script", which allows to configure the simulator for specific scenarios regarding message delivery time, the order in which messages arrive, as well as message and task failures.

6.5 Termination analysis

To also evaluate the practicability of the termination analysis developed in Chapter 3, we implemented a tool in Haskell using the termination criterion based on message type graphs (Corollary 3.15). More precisely, based on the abstract syntax tree provided by our compiler, we implemented the construction of the message type graph and used cyclicity checks on it. Our tool can thus tell that an algorithm always terminates if the termination criterion based on the message type graph applies, that is, there are no cycles in the message type graph. Note that this is still under the assumption that all local actions terminate. If there are cycles in the message type graph, they are output by the tool.

Note that here it can be made use of the idea of a specification language: a full implementation of the algorithm is not necessary to run analyses on it. Our termination analysis does not take the actual Java code into account but is only based on what is relevant to the message type graph: the message types used in the inputs part of input-action pairs and in the `send`/`reply` statements in actions. Thus, while to actually run the algorithm, the Java code has to be complete and adhere to the Java syntax specification, for the termination analysis only the basic frame as specified in Section 6.2 is necessary. Therefore, the analysis can already be run before completing the implementation of an algorithm and the Java compiler is not involved.

6.6 Tests

The implementation is tested with unit tests on different levels. Lexer and parser have their own tests for their specific functionality. Also the termination analysis is tested on example tasks to see whether it finds the correct cycles. In addition, black-box tests take Jial source files as input and check whether after compilation and simulation tasks are in an expected state.

The feedback of different testers with background in computer science has been considered. They provided some additional unit tests which shows that they were able to use the language and increases the implementation's test coverage with independent tests. This can be seen as a first step towards an actual study on the practicability of the language.

6.7 User interface

For the two tools, compiler and termination analysis, we provide a simple command line interface with the following format:

`jialc <msg_type_file> <task_file1> <task_file2> ...`	Compile the given task source files to Java classes.
`jialt <task_file1> <task_file2> ...`	Run the termination analysis on the tasks corresponding to the given source files. Outputs cycles of the message type graph.

6.8 Example

In this section, we want to demonstrate the working of the compiler by implementing the two-phase commit protocol (Algorithm 3.1) in Jial, see how it is translated to Java code and observe its execution in the simulator.

6.8.1 The two-phase commit protocol implemented in Jial

To implement the two-phase commit protocol (Algorithm 3.1) in Jial, we first provide the required message type file (Listing 6.1) and then rewrite the Coordinator and Participant tasks (Listing 6.2 and Listing 6.3).

```
package example;

import example.data.Vote;

abort()
commit()
vote_request()
vote(Vote) // uses the imported type "Vote"
```

Listing 6.1: The message type definitions for the two-phase commit protocol

```
package example;

import example.data.Vote;

task Coordinator {
    private Transaction t;
    private Vote v;
    private int count = 0;

    public Coordinator(Transaction t, Vote v) { // A normal Java constructor can be used
        this.t = t;
        this.v = v;
    }

    input init() {
        if (v == Vote.ABORT) {
```

```
            send abort() to $Participant;
            t.abort();
        } else {
            send vote_request() to $Participant;
        }
    }

    input vote(Vote x) {
        if (x == Vote.ABORT) {
            send abort() to $Participant;
            t.abort();
        } else {
            count++;
            if (count == $Participant.size()) {
                send commit() to $Participant;
                t.commit();
            }
        }
    }
}
```

Listing 6.2: The Coordinator task of the two-phase commit protocol in Jial

```
package example;

import example.data.Vote;

task Participant {
    private Transaction t;
    private Vote v;

    public Participant(Transaction t, Vote v) { // A normal Java constructor can be used
        this.t = t;
        this.v = v;
    }

    input vote_request() {
        reply vote(v);
    }

    input abort() {
        t.abort();
    }

    input commit() {
        t.commit();
    }
}
```

Listing 6.3: The Participant task of the two-phase commit protocol in Jial

6.8.2 Compilation to Java

Listing 6.4 is the message class `M` the compiler creates from the message type definitions in Listing 6.1. Listing 6.5 and Listing 6.6 show the resulting `Coordinator` and `Participant` classes the compiler produces. Comments are added to explain some details. At the very beginning, the special variables containing the IDs of

each task type are generated. Then follows the code of the ⟨*java-declarations*⟩ in the task specification.

The `prepare()` method includes the initialization of the `$Coordinator` and `$Participant` variables as well as the registration of the guard and action methods with the base class.

The remainder of the class definition consists of methods representing the input-action pairs. Each guard and action method starts with the specification of a local variable `$src` to refer to the current message's sender. After this, local variables for the message parameters are defined. Inbound messages are named `_m`, outbound messages `m_`. The extra scopes (`{...}`) ensure that the temporary message variable `m_` can only be accessed by the generated code creating this message.

```
package example;

import base.Message;
import example.data.Vote;

public class M {
    public static class abort extends Message {}

    public static class commit extends Message {}

    public static class vote_request extends Message {}

    public static class vote extends Message {
        public final Vote v;

        public vote(Vote v) {
            this.v = v;
        }
    }
}
```

Listing 6.4: The compiled message type class `M`

```
package example;

import java.util.Set;
import base.Message;
import base.Task;
import example.data.Vote;

public class Coordinator extends Task {
    private Set<Integer> $Coordinator; // Generated special variable
    private Set<Integer> $Participant; // Generated special variable
    private Transaction t;
    private Vote v;
    private int count = 0;

    public Coordinator(Transaction t, Vote v) { // The constructor is simply copied
        this.t = t;
        this.v = v;
    }

    public void prepare() { // Generated initialization method
        super.prepare();
        $Coordinator = getGroup("Coordinator"); // Initialize special variable
        $Participant = getGroup("Participant"); // Initialize special variable
```

```
        // Register input actions pairs (message type, guard, action) with the base class
        addIAP(Message.init.class, m -> init$Guard(m), m -> init$Action(m));
        addIAP(M.vote.class, m -> vote$Guard(m), m -> vote$Action(m));
    }

    public boolean init$Guard(Message _m) {
        int $src = _m.getSrc();
        return true; // no when clause
    }
    public void init$Action(Message _m) {
        int $src = _m.getSrc();
        if (v == Vote.ABORT) {
            { // send abort() to $Participant
                Message m_ = new M.abort();
                m_.setSrc($ID);
                m_.setDest($Participant); // comes from code literally
                send(m_);
            }
            t.abort();
        } else {
            { // send vote_request() to $Participant;
                Message m_ = new M.vote_request();
                m_.setSrc($ID);
                m_.setDest($Participant);
                send(m_);
            }
        }
    }

    public boolean vote$Guard(Message _m) {
        int $src = _m.getSrc();
        Vote x = ((M.vote) _m).v;
        return true; // no when clause
    }
    public void vote$Action(Message _m) {
        int $src = _m.getSrc();
        Vote x = ((M.vote) _m).v; // The parameter of the "vote" message type, named "x"
        if (x == Vote.ABORT) {
            { // send abort() to $Participant;
                Message m_ = new M.abort();
                m_.setSrc($ID);
                m_.setDest($Participant);
                send(m_);
            }
            t.abort();
        } else {
            count++;
            if (count == $Participant.size()) {
                { // send commit() to $Participant;
                    Message m_ = new M.commit();
                    m_.setSrc($ID);
                    m_.setDest($Participant);
                    send(m_);
                }
                t.commit();
            }
        }
    }
}
```

Listing 6.5: The compiled Coordinator task

```
package example;

import java.util.Set;
import base.Message;
import base.Task;
import example.data.Vote;

public class Participant extends Task {
    private Set<Integer> $Coordinator; // Generated special variable
    private Set<Integer> $Participant; // Generated special variable
    private Transaction t;
    private Vote v;

    public Participant(Transaction t, Vote v) { // The constructor is simply copied
        this.t = t;
        this.v = v;
    }

    public void prepare() { // Generated initialization method
        super.prepare();
        $Coordinator = getGroup("Coordinator"); // Initialize special variable
        $Participant = getGroup("Participant"); // Initialize special variable
        // Register input actions pairs (message type, guard, action) with the base class
        addIAP(M.vote_request.class,
                m -> vote_request$Guard(m), m -> vote_request$Action(m));
        addIAP(M.abort.class, m -> abort$Guard(m), m -> abort$Action(m));
        addIAP(M.commit.class, m -> commit$Guard(m), m -> commit$Action(m));
    }

    public boolean vote_request$Guard(Message _m) {
        int $src = _m.getSrc();
        Vote x = ((M.vote) _m).v;
        return true; // no when clause
    }
    public void vote_request$Action(Message _m) {
        int $src = _m.getSrc();
        { // reply vote(v);
            Message m_ = new M.vote(v); // the parameter list is copied form the source
            m_.setSrc($ID);
            m_.setDest($src);
            send(m_);
        }
    }

    public boolean abort$Guard(Message _m) {
        int $src = _m.getSrc();
        return true; // no when clause
    }
    public void abort$Action(Message _m) {
        int $src = _m.getSrc();
        t.abort();
    }

    public boolean commit$Guard(Message _m) {
        int $src = _m.getSrc();
        return true; // no when clause
    }
    public void commit$Action(Message _m) {
        int $src = _m.getSrc();
        t.commit();
    }
}
```

Listing 6.6: The compiled Participant task

6.8.3 Simulation

Running the generated code in the simulator with all tasks voting for "commit" results in the following event log:

```
Coordinator(0) processes init
0 sends vote_request to [1, 2]
Participant(1) processes vote_request from 0
1 sends vote to [0]
Participant(2) processes vote_request from 0
2 sends vote to [0]
Coordinator(0) processes vote from 2
Coordinator(0) processes vote from 1
0 sends commit to [1, 2]
Transaction committed
Participant(1) processes commit from 0
Transaction committed
Participant(2) processes commit from 0
Transaction committed
```

6.9 Practical application

Our implementation can actually be used for real-world applications. Its efficiency mostly depends on that of the Java virtual machine and the implementation of the communication module. If higher performance is required, compilers can be implemented to use different target languages. Other languages would be integrated into the input-action language in the same way as we integrated Java. The parts specific to the base language are pretty small and also the runtime system is not very complicated.

6.10 Conclusion

We showed that the input-action language presented in Chapter 3 can actually be used in practice by extending it to a full programming language and implementing a compiler. The language Jial was obtained by integrating Java into the framework of input-action pairs. Tasks of distributed algorithms can thus be formulated using the full Java language and can be incorporated into regular Java projects.

The language can still be used as a more abstract specification language to run static analysis early on in the design process as the termination analysis is applicable without fully implementing tasks: it suffices to specify the possible message types sent and received. Our implementation of the termination analysis computes the message type graph and outputs its cycles.

Different communication back ends allow to run the compiled tasks on real machines connected by a network or in a simulator on a single machine. The simulator is meant to become the basis for dynamic analyses.

CHAPTER 7

Conclusion

The main goal of this work was to describe a specification language for asynchronous distributed algorithms which allows for static termination analysis. We identified the *input-action language* by Barbosa [Bar96] as a promising candidate and introduced a mathematical model for its semantics.

Making the assumptions of local actions being always terminating and the absence of spontaneous actions in tasks, we were able to reduce the question of termination to communication between tasks. Here the event-driven style of the input-action language, with its input-action pairs providing uniform entry points for control flow, was the key aspect. Given this structure, we were able to develop the message flow graph, representing possible message (and thus global control) flow between the input-action pairs of an algorithm. We identified acyclicity of this graph as a sufficient criterion for termination. However, the restriction of the message flow graph to local properties between two input-action pairs does not allow the reversal of this implication. Nonetheless, we investigated ways of making the termination analysis more precise, by identifying cycles that cannot be traversed completely or which can only be traversed finitely often, which is for example useful for ring algorithms (and allowed us to infer termination for the Chang and Roberts ring algorithm for leader election).

To be able to apply the termination analysis in practice, we approximated the incomputable message flow graph with the efficiently computable *message type graph*. Requiring the specification of a *message type*, the semantic property of a possible message flow between two input-action pairs is reduced to the syntactic property of message types being used when sending and receiving (matching) messages. We argued that the gap between the two graphs is not that relevant in practice and that a proper design often results in them being equal.

Besides termination, we also took a step towards a very different non-functional property, namely fault tolerance. By extending the input-action language with timers, we allowed for the detection of failures using timeouts. As we were able to model timers within the original semantic model, the termination analysis also applies for the extended language and we could thus infer termination for a fault tolerant two-phase commit protocol.

With a second extension, we focused on usability. *Stages* allow us to represent the phases and dynamic behaviour of a protocol. In addition, they allow us to associate timeouts with certain phases of a protocol, making the explicit specification of timers unnecessary and thus ease development.

Finally, we have shown with an implementation that the language and the termination analysis can be applied in practice. The input-action language can

be compiled to Java source code, which together with a runtime system and a communication back end results in executable tasks that can be run on different machines or simulated on a single machine.

Concluding, one can say that we took a step towards showing that specification languages like the input-action language are suitable for static analyses on termination, as one important non-functional property. Regarding our extensions and several suggestions of improvement (for example regarding the termination analysis' precision), it can be said that there is much potential for further research in this area. For example, it would be interesting to consider dynamic systems where tasks can be created at runtime. As we put a lot of focus on distributed protocols, it is also worth investigating how the termination analysis can be complemented with an analysis on *progress*, to show that a protocol will not only terminate but also reach a certain final state. Here our concept of stages can become useful again. Finally, the consideration of other non-functional properties such as fault tolerance deserve further attention.

Bibliography

[Bar96] Valmir C. Barbosa. *An introduction to distributed algorithms*. Cambridge, Massachusetts: MIT Press, 1996.

[CR79] Ernest Chang and Rosemary Roberts. "An Improved Algorithm for Decentralized Extrema-finding in Circular Configurations of Processes". In: *Commun. ACM* 22.5 (May 1979), pp. 281–283. ISSN: 0001-0782.

[Dij75] Edsger W. Dijkstra. "Guarded commands, nondeterminacy and formal derivation of programs". In: *Communications of the ACM* Volume 18 , Issue 8 (Aug. 1975), pp. 453–457.

[Gra78] J. N. Gray. "Notes on data base operating systems". In: *Operating Systems: An Advanced Course*. Ed. by R. Bayer, R. M. Graham and G. Seegmüller. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 393–481. ISBN: 978-3-540-35880-0.

[Hol97] G. J. Holzmann. "The model checker SPIN". In: *IEEE Transactions on Software Engineering* 23.5 (May 1997), pp. 279–295. ISSN: 0098-5589.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. 1st. Morgan Kaufmann, 1996. ISBN: 1558603484.

[Ske81] Dale Skeen. "Nonblocking Commit Protocols". In: *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*. SIGMOD '81. Ann Arbor, Michigan: ACM, 1981, pp. 133–142. ISBN: 0-89791-040-0.

[TV07] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed systems: principles and paradigms*. 4th ed. Upper Saddle River, NJ: Prentice Hall, 2007.

Declaration of Originality

I hereby declare that this thesis is my own original work, which has been composed by me without using aids other than those specified. I have clearly referenced all sources, both published or unpublished, which have been directly or indirectly used in the work. This work has not been submitted, in this or similar form, to any other examination authority.

Ilmenau, 11th December 2018

Felix Wiemuth