Static Termination Analysis for Event-driven Distributed Algorithms

Felix Wiemuth
fw@concordium.com
felix.wiemuth@tu-ilmenau.de
Technische Universität Ilmenau
Ilmenau, Germany

Peter Amthor
peter.amthor@tu-ilmenau.de
Technische Universität Ilmenau
Ilmenau, Germany

Winfried E. Kühnhauser
winfried.kuehnhauser@tu-ilmenau.de
Technische Universität Ilmenau
Ilmenau, Germany

ABSTRACT

Termination is an important non-functional property of distributed algorithms. In an event-driven setting, the interesting aspect of termination is the possibility of control flow loops through communication, which this paper aims to investigate.

In practice, it is often difficult to spot the possible communication behaviour of an algorithm at a glance. With a static analysis, the design process can be supported by visualizing possible flow of messages and give hints on possible sources of non-termination.

We propose a termination analysis for distributed algorithms formulated in an event-driven specification language. The idea is to construct a *message flow graph* describing the possible communication between components (*input-action pairs*). We show that acyclicity of that graph implies termination. While many interesting algorithms indeed contain cycles, we also suggest ways of detecting cycles which cannot lead to non-termination.

As a practical evaluation, we describe a concrete programming language together with a tool for automated termination analysis.

CCS CONCEPTS

• **Theory of computation → Distributed algorithms**; **Program analysis**; • **Software and its engineering → Automated static analysis**; *Specification languages.*

KEYWORDS

distributed algorithms, event-driven, algorithm specification, programming languages, message flow, static analysis, termination

ACM Reference Format:
Felix Wiemuth, Peter Amthor, and Winfried E. Kühnhauser. 2019. Static Termination Analysis for Event-driven Distributed Algorithms. In *DEBS '19: The 13th ACM International Conference on Distributed and Event-based Systems (DEBS '19), June 24–28, 2019, Darmstadt, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3328905.3329500

1 INTRODUCTION

For distributed algorithms, the question of termination is more complex than for local, sequential algorithms. We consider distributed algorithms formulated as a set of local components (tasks), communicating via messages which trigger actions in tasks, resulting in the sending of new messages. For a designer it is important to know whether the concurrent invocation of actions at different tasks eventually comes to a stop. In addition to the question of whether local actions terminate on invocation (which is subject to termination analysis for sequential programs), in a distributed setting there is the possibility of loops through communication which represent another possible source of non-termination. The question is how these loops can be identified and what characteristics the specification language has to provide for this purpose.

As a starting point, we take an event-driven language introduced by Barbosa [1] as a high-level specification language to study the properties of distributed algorithms. It is based on the guarded command language by Dijkstra [3] which allows for the expression of non-determinism – a core characteristic of asynchronous distributed algorithms – by decoupling reception and processing of messages. To this end, actions are preconditioned with the presence of a certain message as well as conditions on a task's local state. For this *input-action language*, we define a semantic model which allows us to represent an algorithm's possible communication as a graph. We then prove that acyclicity of that graph implies termination. As the graph is based on a semantic criterion, it is in general incomputable, and we approximate it using a syntactic criterion.

In practice, the power of our approach lies in the use of an intuitive specification language which allows for an early disclosure of potential communication loops without having to fully implement an algorithm. In the design process of an algorithm, the message flow graph can visualize the possible message flow between an algorithm's components. The analysis basically runs in quadratic time in the number of *input-action pairs* the algorithm consists of.

We identify important classes of distributed algorithms – such as sequential protocols – for which our termination criterion applies. In addition, we discuss how the termination criterion's precision can be improved. The syntactic specification of bounds on the number of invocations of certain actions even allows us to infer termination for a ring algorithm where cycles are traversed repeatedly, but finitely often. From the integration of additional analysis we expect our approach to apply to a wider variety of algorithms. The work is practically evaluated with the implementation of a termination analysis tool as well as a compiler for a Java-based programming language, allowing to produce executable Java bytecode.

For sake of generality, we consider a model for asynchronous distributed algorithms and first assume it to be faultless. However, we also show how it can easily be adapted to represent synchronous execution, failures or even timer-based execution to a certain extent.

DEBS '19, June 24–28, 2019, Darmstadt, Germany

ACM ISBN 978-1-4503-6794-3/19/06…$15.00
https://doi.org/10.1145/3328905.3329500

In summary, the contributions of this work are (1) a formalization of the input-action language's semantics; (2) the concept of a *message flow graph* representing the possible flow of messages between actions of an algorithm; (3) the identification of a termination criterion based on the message flow graph; (4) the application of the termination criterion in practice including an implementation; and (5) ways of improving the termination analysis' precision.

The remainder of this paper is organized as follows. In Section 2, we describe the specification language serving as the basis for our termination analysis and introduce a corresponding semantic model. This model allows us to formalize a termination criterion, prove its correctness and discuss its practical application in Section 3. In Section 4, we take a closer look at the expressivity of our semantic model. We then discuss possible improvements of the termination analysis' precision in Section 5. In Section 6, we describe a compiler for the input-action language and an implementation of a termination analysis tool. Finally, in Section 7 we discuss the termination analysis' applicability before concluding in Section 8.

This paper is based on the first author's Master's thesis [10].

Related work. Several specification languages and tools exist to verify distributed systems. TLA$^+$ ("Temporal Logic of Actions") [6] is a language with an extensive toolchain to verify systems before implementation. The main difference to our approach is its generality. TLA$^+$ (using a declarative notation) can express a variety of properties whereas we focus on a single property (termination) for which we can use the input-action language, allowing for an intuitive imperative specification up to the detail of implementation.

In contrast to TLA$^+$, the verification modelling language Promela is very similar to the input-action language. However, it is again intended to verify a variety of properties with its model checker SPIN, using standard Linear Temporal Logic (LTL) [5].

While the former, very general methods are based on model checking and thus require certain restrictions on the model to be decidable, the focus on termination allows our analysis to be purely static and thus also allows to analyze a priori unbounded systems.

The data flow analysis presented in [9] is based on an *event spanning graph* which connects program flow graphs of each task with message links, similar to our message flow graph. While their analysis can be integrated into ours to improve precision (e.g. by detecting unreachable statements), their graph is also used for more general purposes like narrowing down possible values for variables.

The formalism we introduce has some parallels to *I/O automata*, which are used to formally study asynchronous distributed algorithms in [7]. 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. While I/O automata are more general, our model is simpler and more straight-forward to reason about regarding our particular interest in termination. It also corresponds more directly to the source code of actual programs.

2 EVENT-DRIVEN FORMULATION OF DISTRIBUTED ALGORITHMS

Formulating distributed algorithms in an event-driven style has several advantages over just using, for example, send and receive primitives. In the context of distributed algorithms, the most important event is the reception of a message. Letting an algorithm perform actions only based on the reception of certain messages adds more structure regarding the possible control flow. It is this structure which eventually allows us to establish a termination criterion. For sake of generality, we first assume an asynchronous timing model and later show how it can also be used to apply our analysis to synchronous algorithms.

2.1 The input-action language

A specification language that turned out to be very useful for the formulation of distributed algorithms is the *input-action language* introduced by Barbosa [1]. It is based on the guarded command language by Dijkstra [3], allowing for the expression of non-determinism, a core characteristic of asynchronous distributed algorithms. The concept is also used by Lynch [7] to describe distributed algorithms with I/O automata. The input-action paradigm is not only used in research but also found its practical applications, for example in the verification modelling language Promela [5]. In addition to its practical relevance and its ability to conveniently represent distributed algorithms in a well-readable way, we choose the input-action language because its structure forms the basis for our static analysis.

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, it can, for example, just 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 modifying the local state of a task and possibly sending messages to other tasks. Except for primitives to send messages, the concrete nature of statements is irrelevant to our considerations. Instead, we profit from such a more abstract *specification language* by being able to perform static termination analysis without fully implementing an algorithm.

2.1.1 Semantics. The semantics of the language in an asynchronous timing model 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. This mechanism abstracts from the order in which messages arrive. Furthermore, there is no given order in which the tasks of an algorithm perform actions, which represents full asynchronism.

2.1.2 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 a task is understood 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" [1, 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 [1] actually make use of guards on local state without receiving a message. 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 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 Section 4.1. However, it shall already be noted that by shifting activation to *external messages*, spontaneous actions can be modelled without affecting the termination analysis.

2.1.3 Formulating distributed algorithms in the input-action language. Algorithm 2.1 (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 match the language described so far. It will be formalized in Section 2.1.4.

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" ($P_1, \ldots, P_n$). 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.

Algorithm 2.1 implements the *two-phase commit protocol* (originally described in [4]) 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*). The algorithm is initialized by sending `init()` to the coordinator task.

```
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, ..., 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 2.1: The two-phase commit protocol

To demonstrate the usage of guards in the input-action language, Algorithm 2.2 implements the distributed construction of a list in order. Messages requesting a certain element or the sum of all elements are not processed until they can be answered.

2.1.4 Syntax. In the form of an EBNF grammar, we now formalize the syntax informally introduced before and used throughout all examples. We leave open the concrete form of some elements. Any additional statements can be added to obtain a full programming language from the abstract specification language. For our eventually syntactic termination analysis the described elements are sufficient.

⟨*task*⟩ ::= ⟨*task-name*⟩ ⟨*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*⟩ | ...

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

2.2 Semantic model

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.

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 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.

Definition 2.1. 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 Δ.

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 task only uses message *matchers* in guards 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 2.2. A *task* is a function $g \colon Q \times \mathcal{P}(\mathcal{M}) \to \mathcal{P}(\Delta \times \mathcal{M})$ where for all $q \in Q, M \subseteq \mathcal{M}, \delta \in \Delta$ 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 $\{ \delta \in \Delta \mid \exists q \in Q, m \in \mathcal{M} \colon (\delta, m) \in g(q, \{m\}) \}$ is finite.

A distributed algorithm is then simply a collection of tasks.

Definition 2.3. A *distributed algorithm* is a tuple $\mathcal{A} = (g_1, \ldots, g_n)$ of tasks g_i for $i \in I$, also written as $(g_i)_{i \in I}$.

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 2.4. 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 2.5. The *successor relation* $\to_{\mathcal{A}} \subseteq C \times C$ for $\mathcal{A}$ is defined by

$$
\begin{aligned}
&((q_1, M_1), \ldots, (q_k, M_k), \ldots, (q_n, M_n)) \\
&\to_{\mathcal{A}} ((q_1, M_1'), \ldots, (q_k', M_k'), \ldots, (q_n, M_n')) \\
\iff\ &\exists k \in I, (\delta, m) \in g_k(q_k, M_k) \colon \delta(q_k, m) = (q_k{}', M_1^+, \ldots, M_n^+) \\
&\text{where } M_i' = \begin{cases} (M_i \setminus \{ m \}) \cup M_i^+ & \text{if } i = k \\ M_i \cup M_i^+ & \text{if } i \neq k \end{cases}.
\end{aligned}
$$

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 TERMINATION ANALYSIS

We now develop a termination criterion for distributed algorithms formulated in the semantic model introduced in the previous section. To focus on the interesting aspect of distributed algorithms (their

global behaviour through communication) 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.

From now on, we consider an arbitrary algorithm $\mathcal{A}$ with n tasks and fixed state and message sets Q and $\mathcal{M}$, 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.

3.1 A termination criterion

Before describing the idea of the termination criterion, we precisely define what it means for an algorithm to terminate in terms of the semantic model. Assuming that messages are delivered after a finite amount of time and local actions to terminate, 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 2.1) 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 3.1. 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.

3.2 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 any 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 3.2. The *message flow graph* for $\mathcal{A} = (g_i)_{i\in I}$ is the directed graph $G(\mathcal{A}) = (\Delta, E)$ where for all $\delta, \delta^* \in \Delta, k \in I, c = (q_i, M_i)_{i\in I}, c' = (q_i', M_i')_{i\in I} \in C, m \in M_k$ 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 2.5:

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

Fig. 1 shows the message flow graph for the two-phase commit protocol (Algorithm 2.1). The actions are simply labelled 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. 2 (that is, here we merge $\mathrm{P}_1, \ldots, \mathrm{P}_n$ into P).

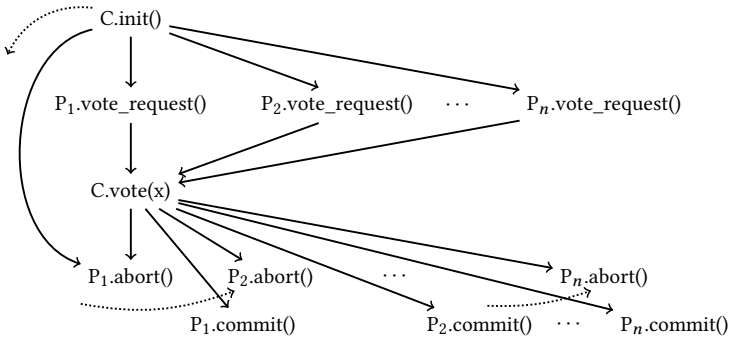


Figure 1: The message flow graph of Algorithm 2.1

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. 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 (process) 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

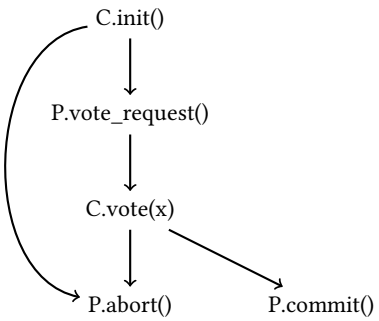


Figure 2: The simplified message flow graph of Algorithm 2.1

and the newly created messages are associated with edges as described above. The association of messages to edges can change whenever 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 3.3. *If $G(\mathcal{A})$ is acyclic, then $\mathcal{A}$ is terminating on all configurations.*

According to this criterion, the two-phase commit protocol (Algorithm 2.1) always terminates, as its message flow graph (Fig. 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, in Section 3.4 we show how it can be used in practice anyway.

First however, we take a look at the termination criterion's inherent limitation: equivalence of acyclicity and termination can not 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 reachable from configurations where δ_2 can process a message from δ_1 could be disjoint from the configurations where δ_2 can send a message to δ_3. 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 5.

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 processed 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 processed 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 the actions of an acyclic message flow graph using *ranks*, where the rank of an action denotes the maximum length of a sequence of actions subsequently invoked.

Definition 3.4. Let $G = (\Delta, E)$ be an acyclic directed graph on actions. The *rank* $r_G(\delta)$ of an action δ in G is the length of a maximal path originating at δ (or 0 if δ has no outgoing edges).

Messages can then only flow to actions with a lower rank:

Lemma 3.5. *Let $G = (\Delta, E)$ be an acyclic directed graph on actions. Then $(\delta, \delta^*) \in E$ implies $r_G(\delta) > r_G(\delta^*)$.* □

To prove that the algorithm will indeed terminate, imagine the execution of actions to cost *fuel*. If we set the cost for one action to be executed at one unit of fuel and provide the algorithm with a finite amount of fuel, the number of actions executed in total is limited by the given amount of fuel. To make sure that whenever an action is to be executed the fuel to do so is available, we let each message "carry" at least as many units of fuel as might be required for the execution of all actions caused by this and the recursively created messages.

An action of rank 0 can produce no messages which can later be processed by an other action. Therefore, messages sent to actions of rank 0 do not have to carry any extra fuel for further activations. An action of rank 1 can produce at most $b - 1$ messages (b is the constant we chose to be greater than the maximum number of messages sent by any action). As by Lemma 3.5 all actions that can process a message from an action of rank 1 must have rank 0, $b - 1$ is also the maximum number of actions recursively caused to be executed by those messages. Thus, a message sent to an action of rank at most 1 should carry at least $b - 1$ units of extra fuel. In general, the number of actions recursively activated by an action δ is less than b to the power of $r_G(\delta)$. Consequently, an amount of $b^{r_G(\delta)}$ units of fuel is sufficient to execute δ and provide fuel for all recursive activations.

To formalize this idea, each message in an input buffer is assigned a *cost* equivalent to an amount of fuel that is sufficient for the execution of all actions recursively activated by that message. As a message can possibly be processed by different actions, we have to provide enough fuel for the action with the highest rank.

The cost of a message m (in input buffer M_k) in a certain configuration is thus defined as b to the power of the maximum rank of an action (of task k) which can process m in the current or any future configuration.

Definition 3.6. For a configuration $c = (q_i, M_i)_{i\in I} \in C$ and $k \in I$ we define the *cost* of $m \in M_k$ by

$$y_k^G(c,m) = b^{r_G^{\max}(k,c,m)} \text{ where } \qquad \text{(defining } \max(\emptyset) = 0\text{)}$$
$$r_G^{\max}(k,c,m) = \max\{\, r_G(\delta^*) \mid \exists c^* = (q_i^*, M_i^*)_{i\in I} \in C\colon c \to_{\mathcal{A}}^* c^* \ \wedge\ (\delta^*, m) \in g_k(q_k^*, M_k^*)\,\}.$$

The following lemma is the key of the proof. It shows that the costs of all messages created by an action can be covered by the cost of a message invoking it. This is achieved by showing that the cost of each message created is less than a $\frac{1}{b-1}$ of the consumed message's cost.

Lemma 3.7. *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 2.5. If $G(\mathcal{A}) = (\Delta, E)$ is acyclic then $y_k^G(c,m) > (b-1)y_j^G(c', m')$ for all $j \in I$ and $m' \in M_j^+$.*

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 $\delta^* \in \Delta$ 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:

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

With

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

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

$$r_G^{\max}(k,c,m) > r_G^{\max}(j, c', m')$$

and thus (as $b^x > (b-1)b^{x-1}$ for all $x \geq 1$)

$$y_k^G(c,m) = b^{r_G^{\max}(k,c,m)} > (b-1)b^{r_G^{\max}(j,c',m')} = (b-1)y_j^G(c', m')\ .$$

□

To complete the proof, it only remains to show that in each step the combined cost of all messages in the input buffers decreases, corresponding to the consumption of fuel for the action executed. To this end, we define the *value* of a configuration to be the sum of the costs of all messages in the input buffers.

Definition 3.8. The *configuration value* of a configuration $c = (q_i, M_i)_{i\in I} \in C$ is $v_G(c) = \sum_{i\in I} \sum_{m\in M_i} y_i^G(c,m)$.

Lemma 3.9. *Let $c, c' \in C$ with $c \to_{\mathcal{A}} c'$. If $G(\mathcal{A})$ is acyclic then $v_G(c) > v_G(c')$.*

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 2.5. Then

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

With $\sum_{j\in I} |M_j^+| < b$ and, by Lemma 3.7, $y_k^G(c,m) > (b-1)y_j^G(c', m')$ for all $j \in I$ and $m' \in M_j^+$ we obtain the claimed inequality. □

Finally, we can prove Theorem 3.3.

Proof (Theorem 3.3). Let $G(\mathcal{A})$ be acyclic 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 3.9 $v_G(c_0) > v_G(c_1) > \cdots > v_G(c_\ell)$. As $v_G(c) \geq 0$ for all $c \in C$, we obtain $\ell \leq v_G(c_0)$. Thus, every sequence of successor configurations must be finite, i.e., $\mathcal{A}$ is terminating. □

3.4 The termination criterion in practice

There is one missing part in turning the termination criterion into a termination analysis tool: we have not discussed yet how we can make use of the incomputable message flow graph in practice. The solution is overapproximation, that is, the use of graphs containing the message flow graph. To this end, we first generalize Theorem 3.3.

Corollary 3.10. *Let G be an acyclic directed graph that contains $G(\mathcal{A})$. Then $\mathcal{A}$ is terminating on all configurations.* □

The goal of this section is to find a practically useful approximation of the message flow graph. To this end, we have to identify a meaningful 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 is to base it on purely syntactic properties of an algorithm's textual representation which imply the desired semantic properties. To this end, 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 2.1.4.

3.4.1 The message type graph. A quite natural and efficiently computable syntactic property which can be used to approximate the message flow graph is 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)`. When sending a message, the message type together with values for the parameters has to be specified. 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. We have actually used this kind of message types in Algorithm 2.2 and also our other examples use message types (they just do not specify types for the parameters).

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 3.11. The *message type graph* for an algorithm $\mathcal{A}$ (given in a syntax using message types) is the directed graph $G_T(\mathcal{A}) = (\Delta, E)$ 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 3.12. *$G_T(\mathcal{A})$ contains $G(\mathcal{A})$.* □

With Corollary 3.10, we can formulate the termination criterion in terms of message type graphs.

Corollary 3.13. *If $G_T(\mathcal{A})$ is acyclic, then $\mathcal{A}$ is terminating on all configurations.* □

3.4.2 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 2.1, the message type graph is actually identical to the message flow graph (Fig. 2). The reason is that each message flow included in the message type graph is actually possible.

For illustration, we now also construct an example where the message type graph differs from the message flow graph. Algorithm 3.1 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 3.1: Impossible message flow – the message flow graph (middle) has no edges, the message type graph (right) has a loop

We can identify multiple reasons why a message flow in the message type graph is actually not possible. 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 possibilities of improvement, standard techniques for sequential program analysis can be used (see for example [8]). 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 could be used to reveal design mistakes.

3.4.3 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 3.13 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 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.

4 DISCUSSION OF THE SEMANTIC MODEL

In this section, we review our semantic model by first coming back to the restriction regarding spontaneous actions and then discussing what algorithms and behaviour can be represented by the model.

4.1 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. In the presence of spontaneous actions, the message flow graph would be of no use as actions could occur again at any time.

Nonetheless, spontaneous actions play an important role in practice, for example in the form of timers which allow algorithms to perform actions after a timeout. We therefore discuss how timers can still be realized.

Simulating spontaneous actions and timers. As in our model actions can only be performed by consuming a message, spontaneous actions are not supported directly. However, they can be simulated by adding dummy messages to input buffers, triggering certain actions. These dummy messages can be provided either internally via `send` statements or externally. Starting a timer is then simulated by sending a timeout message which can later be handled by a "timeout action". Note that durations are not modelled and the simulated timeout can occur at any time relative to other steps in the execution. Loops arising through this construction represent the possibility of control-flow loops through timers which of course exist if actions can repeatedly invoke each other. Should this repetition in fact be limited, these loops can be tackled with the proposals of improving precision in Section 5. An example of using timers (a fault tolerant two-phase commit protocol) can be found in [10].

External messages as an alternative. To allow for repeated spontaneous actions in the described simulation, the dummy messages have to be restored by the spontaneous actions themselves, leading to loops in the message flow graph. This renders the termination analysis useless. An alternative to internally providing the dummy messages is to allow each spontaneous action externally by modifying an algorithm's configuration at any point during its execution, adding a dummy message. This way 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 3.3 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 still terminates.

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, without the need to externally modify configurations. 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.

4.2 Model variants

While we only described a model for asynchronous distributed algorithms in an ideal environment, it can easily be modified to represent other behaviour. If the model is only modified by removing transitions from the successor relation, the termination criterion, which holds for all executions of the original model, applies as before. Also the addition of transitions does not affect the proof as long as they satisfy Lemma 3.9 (configuration values decrease with each step). This allows for several interesting model variants.

Synchronous execution, where messages are delivered and processed in lock step, is achieved by annotating messages with the current round number, allowing to only process messages from the previous round and increasing the round number when all messages from the current round have been processed. Practically, rounds can be initiated by timers or controlled via a global coordinator.

Message loss is modelled by transitions which remove messages from input buffers, reducing the configuration value. Task failure (also temporary) can be modelled by discarding messages sent to the respective task and not letting the task take steps.

5 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 Section 3.4.2, 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 explore other ways of making the termination analysis more precise.

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 3.2). We present two approaches of getting around this problem. The first is about identifying cycles which actually cannot even be traversed as a whole and the second about such that can only be traversed finitely often. To start with, we take a closer look at the limitations of the message flow graph.

The message flow graph's imprecision is due to its inability of capturing message flow over sequences of configurations, 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). First of all, recall from Section 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).

5.1 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 consider the dependencies between these sets: each configuration in $C_{2/3}$ has to be reachable from at least one configuration in $C_{1/2}$ and so on. 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.

A concrete way of finding such impossible message flows is through the use of data flow analysis [9]. It allows for example to detect unreachable statements and narrow down possible values for variables and messages over whole distributed executions.

To address the second limitation of the message flow graph, the set $C_{1/2}$ in the above approach can be restricted to configuration reachable from initial configurations.

5.2 Limited edges

Another 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.

We now come 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 of a limit is of course undecidable in general. However, there are also simple but still interesting cases where the detection is straight-forward, for example if the invalidation of a guard after a certain number of activations is evident (e.g. by checking the value of a counter). 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.

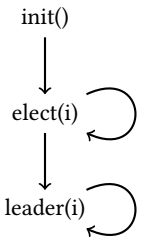
5.3 Inferring termination for a ring algorithm

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.

The *Chang and Roberts algorithm* (Algorithm 5.1) [2] implements leader election on a ring topology. 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* and merging the n copies can be done by multiplying the limit with n). Unrolling the message flow graph yields an acyclic graph (Fig. 3). The corresponding syntactic transformation of Algorithm 5.1 results in Algorithm 5.2. 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 is actually the message flow graph of Algorithm 5.2, 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 5.1: 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.

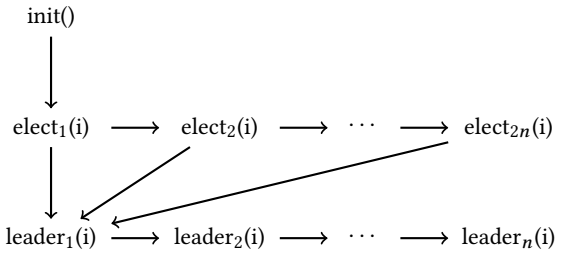


Figure 3: The unrolled message flow graph of Algorithm 5.1

6 IMPLEMENTATION

To demonstrate our concepts, we now describe a programming language based on the abstract input-action language. We then describe how this language can be compiled to executable code and how we implemented a tool for automatic termination analysis. The Haskell source code of the tools is available online[1].

A language which can be compiled to executable code can be obtained by refining the syntax from Section 2.1.4. We do so by integrating a general-purpose imperative language into the specified framework. We choose the commonly used, highly-portable

[1]https://github.com/felixwiemuth/JIAL

```
Ring node
Variables
    leader // the current leader
    self // own ID
    next // the next task on the ring
input init() {
    send elect1(self) to next
}
input elect1(i) {
    if i = self then
        send leader1(i) to next
    else
        send elect2(max(i, self)) to next
    end
}
input elect2(i) {
    if i = self then
        send leader1(i) to next
    else
        send elect3(max(i, self)) to next
    end
}
...
input elect2n(i) {
    if i = self then
        send leader1(i) to next
    else
        error "Limit exceeded"
    end
}
input leader1(i) {
    leader := i
    if i ≠ self then
        send leader2(i) to next
    end
}
input leader2(i) {
    leader := i
    if i ≠ self then
        send leader3(i) to next
    end
}
...
input leadern(i) {
    leader := i
    if i ≠ self then
        error "Limit exceeded"
    end
}
```

Algorithm 5.2: The Chang and Roberts ring algorithm for leader election, transformed according to the unrolled message flow graph (Fig. 3)

Java language for this purpose. We call the resulting language Jial, which is short for *Java input-action language*.

A task in Jial is very similar to a Java class: it can contain arbitrary Java declarations (in the ⟨*variable-declarations*⟩ part) and its logic is described by the method-like input-action pairs. As before, an input-action pair consists of a message matcher, an optional guard and a sequence of statements. As in our examples, a ⟨*msg-matcher*⟩ is realized via the specification of a message type, in the form of type signatures: a name with a list of parameters. A guard simply consists of a Boolean expression on the message parameters and task variables. The body (action) of an input-action pair consists of arbitrary Java statements, with access to the message parameters and task variables. Messages for the `send` and `reply` statements are constructed by specifying the message type and a list of parameters in the form of Java expressions. Task IDs are integers, where destinations can be specified using a `Set<Integer>`. Some special variables are available in guards and actions to access the metadata of the current message (such as its origin) as well as predefined sets of task IDs (e.g. for all instances of a task). In Listing 1, `$Coordinator` denotes the set of IDs of Coordinator tasks and `$src` the origin of the message in consideration. Using this in the `when` clause makes sure that only the coordinator can make participants abort or commit.

```
package example;
task Participant {
    private Transaction t; // Can use classes from same package or use imports
    private Vote v;
    public Participant(Transaction t, Vote v) { // A normal Java constructor
        this.t = t;
        this.v = v;
    }
    input vote_request() { reply vote(v); }
    input abort() when $Coordinator.contains($src) { t.abort(); }
    input commit() when $Coordinator.contains($src) { t.commit(); }
}
```

Listing 1: The Participant task of Algorithm 2.1 in Jial

The similarity of tasks in Jial to Java classes makes it easy to compile a task to an actual Java class. The compiler only has to detect the structure of input-action pairs and generate code for the sending of messages in actions while Java code can be kept as-is. A small runtime system organizes the sending and reception of messages as well as the selection of executable actions. The implementation of a communication module, providing the sending of messages via task IDs over a network, is left to the application. With a communication module just simulating network communication, algorithms can be simulated on a single machine. This forms the basis of complementing the static analysis with a dynamic analysis.

Based on our syntactic termination criterion (Corollary 3.13), we complement our compiler with a termination analysis tool. It constructs the message type graph from the abstract syntax tree provided by the compiler and checks it for cyclicity. If there are no cycles, our tool can thus tell that the algorithm will always terminate under the assumption that local actions terminate. If there are cycles in the message type graph, they are output by the tool. Note that to run the tool, no full implementation of the algorithm is required – only the elements specified in the more abstract specification language (Section 2.1.4) are considered.

7 APPLICABILITY

In this section we discuss the practical applicability of our analysis by looking at a number of qualitative aspects of the analysis and its applicability to different classes of algorithms.

The basic termination analysis described in Section 3 reliably infers termination for algorithms with acyclic message flow graphs. This already covers interesting classes of algorithms such as sequential protocols. Such protocols can still be complex and acyclicity might not be obvious during design and specification.

However, as soon as there is a cycle, our analysis can make no claim about termination. It does, though, still provide the message type graph. Visualization of message flow is a useful tool in itself as it can for example reveal unintended message flows. Furthermore, even if cycles cannot be avoided, they can at least be located and special care can be taken to ensure the desired behaviour.

Nonetheless, we still want to analyze termination for algorithms which contain cycles. To this end, we discussed possibilities of identifying cycles which cannot lead to non-termination, improving the analysis' precision. Especially the integration of other static analysis, for local actions as well as global data flow analysis, is a promising opportunity.

Taking visualization a step further, our analysis could be used as a form of graphical proof assistant, visualizing message flow while writing a specification, allowing for iterative rewriting. Especially with the integration of further static analysis, this can be used to find a specification where termination can be inferred.

Regarding that termination is undecidable in general and decidable cases are hard to tackle, we think that our approach, being open to different kinds of improvement, can be a useful complement to existing means. The relation to other approaches (e.g. [6], [5], [9]) regarding for which classes of algorithms, including real-world applications, termination can be inferred is still an open question subject to future work.

8 CONCLUSION

For an event-driven specification language for distributed algorithms, the input-action language, we developed a static termination analysis based on an algorithm's communication behaviour.

Current approaches to statically analyze termination are mostly based on general verification using temporal logics. We complemented these methods by a technique focusing on termination. This allowed us to use the practical input-action language, the abstractions of which form the basis of our analysis. We introduced the concept of a *message flow graph*, representing possible message (and thus global control) flow between the input-action pairs of an algorithm. Assuming that local actions terminate on each invocation, we proved that acyclicity of this graph implies termination.

As the message flow graph with its semantic definition is in general not computable, we introduced the *message type graph*, an efficiently computable approximation of the message flow graph. We then discussed how the termination analysis' precision can be improved, for example by the integration of other static analyses.

Finally, we described the programming language Jial, obtained by integrating Java into the input-action language. It compiles to Java and enables simulation, forming the basis of a complementary dynamic analysis. Our termination analysis tool applies to Jial.

The next step is to make the analysis applicable to a wider range of algorithms, using our proposals of improving precision.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and our shepherd Alessandro Margara for their valuable feedback. We further would like to thank Concordium Research ApS for supporting this work.

REFERENCES

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

[2] Ernest Chang and Rosemary Roberts. 1979. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM*, 22, 5, (May 1979), 281–283. DOI: 10.1145/359104.359108.

[3] Edsger W. Dijkstra. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, Volume 18 , Issue 8, (August 1975), 453–457.

[4] R. Bayer, R. M. Graham, and G. Seegmüller, editors. 1978. *Notes on data base operating systems. Operating Systems: An Advanced Course*. Springer Berlin Heidelberg, Berlin, Heidelberg, 393–481. DOI: 10.1007/3-540-08755-9_9.

[5] G. J. Holzmann. 1997. The model checker spin. *IEEE Transactions on Software Engineering*, 23, 5, (May 1997), 279–295. DOI: 10.1109/32.588521.

[6] Leslie Lamport. 1999. Specifying concurrent systems with tla+. *Calculational System Design*, (April 1999), 183–247.

[7] Nancy A. Lynch. 1996. *Distributed Algorithms*. (1st edition). Morgan Kaufmann.

[8] Flemming Nielson, Chris Hankin, and Hanne R. Nielson. 2005. *Principles of program analysis*. (Corrected edition). Springer, (December 2005).

[9] John H. Reif and Scott A. Smolka. 1990. Data flow analysis of distributed communicating processes. *International Journal of Parallel Programming*, 19, 1, (February 1990), 1–30. DOI: 10.1007/BF01407862.

[10] Felix Wiemuth. 2018. A specification language for distributed algorithms. Master's Thesis, Technische Universität Ilmenau. (December 2018). DOI: 10.22032/dbt.38242.