DaProof

Hans-Dieter A. Hiep[*]
#10196951

July 26, 2017

1 Introduction

This paper reports on the study of a foundational modeling of distributed algorithms, called DaProof. This study was supervised by Alban Ponse (June 2017–July 2017). The work described herein builds further upon an earlier project by the author, DaViz supervised by Wan Fokkink (November 2016–May 2017). DaViz is a software tool for exploring executions of distributed algorithms (hence the "Da"-prefix of both project names). DaProof is motivated by two concerns:

1. In DaViz, distributed algorithms need to be explicitly encoded into a framework. In DaProof, a central theme is developing a model of the underlying structures of this framework. Do these structures reveal useful decompositions of distributed algorithms?
2. Some distributed algorithms are published without a formalized c.q. certified proof of correctness[1], possibly leading to incorrect proofs[3]. What is necessary of a formal model by which properties of distributed algorithms can be proven correct?

The first concern is practical in nature: decomposing existing distributed algorithms into more elementary ones may ease encoding such algorithms in the framework. Decomposition is also useful when dealing with correctness, e.g. showing that the composition of two correct algorithms lead to another correct algorithm. The second concern is related to verification; what properties can be shown, by construction, to hold? These two concerns are interrelated, and we will see how.

This document is structured to document the findings relating these two concerns. First, in section 2, we develop the background material for looking at executions as partially commutative monoids. In section 3, we develop a formal model of distributed algorithms and their executions. In section 4, we discuss the many different assumptions and how these relate to the formal model proposed a section earlier. In section 5, we see an encoding of Cidon's distributed depth-first search algorithm. This paper concludes in section 6, that gives an answer to the two concerns above.

The author would like to thank Alban Ponse for his supervision, critical questions and proof reading, and Jasmin Blanchette for useful suggestions during formalizing.

[*] As part of an Individual Project in the pursuit of a masters degree in Computer Science

2 About Monoids

In this section we will study (many) monoids and how they are related to concurrent processes. This is done from the perspective of an interactive theorem prover, such as Coq[2], that builds proof terms from inductive data structures. As it turns out, not every structure can be modeled precisely by inductive data structures: for some structures we need to introduce a different notion of equality to identify two otherwise unequal elements, and carry around a custom decision procedure for this equality.

Our guiding motto for this section is that of representing abstract structures. We can think of abstract data structures as a structure of which concrete data structures are instances. We will see free constructions of abstract structures, which can be seen as a way to lazily capture the applications of the operations on any concrete instance. If from the laws of an abstract data structure we can derive theorems, we know that such theorems also apply to any concrete instance. Hence these theorems can also be applied up front to reason about any free construction of the abstract data structure.

2.1 Monoids

Inductive data structures can be defined as term languages. For some problems, term languages precisely capture the problem domain. By defining structures as term languages, syntactical equality of terms is equality of elements in an inductive data structure. On the other hand are abstract data structures. Abstract data structures specify some sets, the structure between elements, and an equational specification.

We first develop monoids in a rather verbose manner, to show the same development later we we deal with partially commutative monoids. We define monoids as the following abstract data structure:

Definition 1. A *monoid* $\mathbb{M}(S, \cdot, 1)$ is:

1. a set S
2. a binary operation $\cdot : S \times S \to S$
3. a unit element $1 : S$

such that:

1. associativity $\forall s, t, u \in S. \quad (s \cdot t) \cdot u = s \cdot (t \cdot u)$
2. identity $\forall s \in S. \quad 1 \cdot s = s = s \cdot 1$

In the above definition, the first part is called the *signature* and the second part the *equational specification*. One can think of the signature as an existential quantification, specifying the least requirements of a structure, and think of the equational specification as a universal quantification that must hold for all elements in the structure.

Note that we implicitly require some notion of equality in the equational specification: this equality must be a congruence relation over the signature. Recall that congruence relations are equivalence relations (reflexive, symmetric, transitive) such that the structure is preserved. In the case of monoids, we have that

$$x = y \text{ and } z = w \quad \Rightarrow \quad x \cdot z = y \cdot w$$

We define the free monoid using the following inductive data structure. Let $\mathscr{T}(C)$ be the term language freely generated by the signature, generated by some fixed carrier set C. In other words, $\mathscr{T}(C)$ is the smallest set such that:

1. it is generated by C, i.e. $C \subseteq \mathscr{T}(C)$
2. it is closed under binary operation $\cdot : \mathscr{T}(C) \times \mathscr{T}(C) \to \mathscr{T}(C)$
3. it is closed under unit $1 : \mathscr{T}(C)$

If we take syntactical equivalence as equality in the monoid, this term language is not a monoid since the monoid laws do not hold, e.g. $x \cdot 1 \neq x$ or $(x \cdot y) \cdot z \neq x \cdot (y \cdot z)$. We also define the following congruence relation. Let $\cong$ be the smallest congruence relation on $\mathscr{T}(C)$, such that:

1. it is associative, $\forall x, y, z \in \mathscr{T}(C). \quad (x \cdot y) \cdot z \cong x \cdot (y \cdot z)$
2. it respects identity $\forall x, y, z \in \mathscr{T}(C). \quad 1 \cdot x \cong x \cong x \cdot 1$

The monoid laws. We now define the free monoid by the following quotient structure.

Definition 2. Let C^* be the *free monoid* generated by C, that is $\mathscr{T}(C)/\cong$.

The elements of C^* are the equivalence classes $[x] = \{y \in \mathscr{T}(C) \mid y \cong x\}$. This structure is not a good concrete representation, each equivalence class is an infinite set of terms: x, $x \cdot 1$, $x \cdot (1 \cdot 1)$, et cetera. We will need a decision procedure for deciding equivalence, from which we then choose unique representatives of the equivalence class. These unique representatives are then selected as elements of the free monoid, where equivalence is decided by normalizing to the selected representatives.

We take the following (typical) normal form as unique representatives:

$$1 \quad x \cdot 1 \quad y \cdot (x \cdot 1) \quad z \cdot (y \cdot (x \cdot 1)) \quad \ldots$$

We choose to order the associativity law from left to right (i.e. right-associative), but this choice is arbitrary. The elements of the carrier set are not in normal form; instead, we consider terms consisting of the structure $(x \cdot xs)$ where x is an element of the carrier set and xs is another normalized term. In other words, we say a term in $\mathscr{T}(C)$ is in normal form if:

1. the term is unit
2. the term is a binary operation $(x \cdot xs)$ such that $x \in C$ and xs in normal form

We can precisely capture these unique representatives in a term language.

Definition 3. Let the *free monoid* C^* generated by C be the smallest set such that:

1. the unit $\varepsilon \in C^*$
2. the constructor ("cons") $x \# xs \in C^*$ where $x \in C$ and $xs \in C^*$

This definition corresponds with the usual definition of polymorphic lists. We also have a normalization procedure that, given any term in $\mathscr{T}(C)$, results in the unique representative of the equivalence class of the given term.

$$\begin{aligned} @ &: C^* \to C^* \\ \varepsilon \mathbin{@} x &= x \\ (x\#y) \mathbin{@} z &= x\#(y@z) \end{aligned}$$

$$\begin{aligned}
\pi &: \mathscr{T}(C) \to C^* \\
\pi(1) &= \varepsilon \\
\pi(x) &= x\#\varepsilon \\
\pi(x \cdot y) &= \pi(x)@\pi(y)
\end{aligned}$$

We can show that for @ ("append"), given two normal forms, the result is also in normal form. Hence by induction on $\mathscr{T}(C)$, π indeed is a normalization procedure. The free monoid as given is also a monoid: take ε as unit and @ as binary operator. As can be seen from π, the free monoid is generated by mapping $x \in C$ to $x\#\varepsilon \in C^*$. Showing the monoid laws is easy. We also need the representation function:

$$\begin{aligned}
\rho &: C^* \to \mathscr{T}(C) \\
\rho(\varepsilon) &= 1 \\
\rho(x\#xs) &= x \cdot \rho(xs)
\end{aligned}$$

We have that $\rho(\pi(x)) \cong x$ holds, i.e. normalization preserves our notion of equality. For quotient types, we furthermore require $\pi(\rho(x)) = x$, as is explained in [4].

2.2 Monoid Overview

$$\begin{tikzcd}
\mathbb{M} \arrow[d] & \mathbb{PCM} \arrow[d] & \mathbb{CM} \arrow[d] & \mathbb{ICM} \arrow[d] \\
C^* \arrow[d, leftrightarrow] & C^{I*} \arrow[d, leftrightarrow] & \mathbb{N}^C \arrow[d, leftrightarrow] & \mathscr{P}(C) \arrow[d, leftrightarrow] \\
\text{sequence} \arrow[d] & \text{trace} \arrow[d] & \text{multiset} \arrow[d] & \text{set} \arrow[d] \\
\text{list} & +\text{toposort} & +\text{sorted} & +\text{unique}
\end{tikzcd}$$

Figure 1: From left to right, we have monoids ($\mathbb{M}$), partially commutative monoids ($\mathbb{PCM}$), commutative monoids ($\mathbb{CM}$), idempotent commutative monoids ($\mathbb{ICM}$).

There are many different monoids where we require more laws to hold, which we can depict as in Figure 1. The monoids in this picture are ordered from most general to least general, in the following sense: every partially commutative monoid is also a monoid, and every commutative monoid is also a partially commutative monoid, and so on. We give definitions for each of the new monoids.

Definition 4. A *commutative monoid* $\mathbb{CM}(S, \cdot, 1)$ is a monoid $\mathbb{M}(S, \cdot, 1)$ such that:

1. commutativity $\forall s,t \in S. \quad s \cdot t = t \cdot s$

An *idempotent commutative monoid* $\mathbb{ICM}(S, \cdot, 1)$ is in addition:

2. idempotent $\forall s \in S. \quad s \cdot s = s$

Before continuing with partially commutative monoids, we observe that finding normal forms for (idempotent) commutative monoids is not trivial. Indeed, we cannot order the commutative law in any way that results in a terminating rewrite system, e.g. $x \cdot y = y \cdot x = x \cdot y = \ldots$.

For a concrete data structure that represents the free constructions of these monoids, we require in addition a relation to hold between elements of the carrier set: a decidable, strict total order. Since we do not want to prove that every carrier set has such a relation, this might not be a free construction. However, for our purposes where we restrict ourselves to carriers where such a relation always exists.

We will call the free constructions as follows: traces are free partially commutative monoids (FPCM), multisets are free commutative monoids (FCM) and sets are free idempotent commutative monoids (FICM). Since we require the carriers to be a sets, the notion that the idempotent commutative monoid freely generated by some set might seem circular (i.e. "sets" are freely generated by sets); but we will not concern ourselves with FICMs here and given them only for illustration purposes.

2.3 Partially commutative monoids

We define partially commutative monoids and their free construction in a similar way as before. This is done similar to others[8]. We first define the notion of independence relation.

Definition 5. An *independence relation* $I \subseteq S \times S$ is an irreflexive, symmetric relation over S.

This notion corresponds to concurrency in distributed algorithms. We will use examples from concurrency to explain partially commutative monoids. We define partially commutative monoids as follows.

Definition 6. A *partially commutative monoid* $\mathbb{PCM}(S, I, \cdot, 1)$ is:

1. a set S
2. an independence relation $I \subseteq S \times S$
3. a binary operation $\cdot : S \times S \to S$
4. a unit element $1 : S$

such that:

1. associativity $\forall s,t,u \in S. \quad (s \cdot t) \cdot u = s \cdot (t \cdot u)$
2. identity $\forall s \in S. \quad 1 \cdot s = s = s \cdot 1$
3. partial commutation $\forall s,t \in S. \quad (s,t) \in I \Rightarrow s \cdot t = t \cdot s$

Compare this definition to Definition 4: the equational specification of commutativity is now conditioned by the independence relation. A partially commutative monoid (PCM) with a maximal independence relation (i.e. $S \times S \backslash \{(x,x) \mid x \in S\}$) is also a commutative monoid (CM), since the partial commutation law must hold for all distinct elements $s,t \in S$. Conversely, every CM is also a PCM for any independence relation, since every two elements already commute in CM.

For example, take a linearization of events happening at multiple nodes. Two events are concurrent if they are not related by a causal relation. Hence, two linearizations are behaviorally equivalent if one is the result of swapping the two concurrent events in the other. The notion of all equivalent linearizations is called a computation, defined to be the equivalence class of linearizations that are behaviorally equivalent. If we take all linearizations as elements, behavioral equivalence as the independence relation between linearizations, concatenation of linearizations as binary operator and the empty linearization as unit, then we obtain another partially commutative monoid.

We again show the construction of the free partially commutative monoid, that is the partially commutative monoid freely generated by some fixed carrier set C. In our running example, this freely generated PCM is interesting since we do not have to deal with linearizations but instead with events and the notion of concurrency directly. We have two choices to define such structure:

1. We again take a term language and take the quotient over the smallest congruence relation consisting of all laws
2. Since a commutative monoid is also a monoid, we take the free monoid and take the quotient over an equivalence relation that captures the partial commutation law

We take the first. The second is explored by others[6]: in that case we have the least congruence $\sim$ such that $s \cdot t \sim t \cdot s$ for $(s,t) \in I$ that is used to quotient C^*. We again have the term language $\mathscr{T}(C)$ consisting of $C \subseteq \mathscr{T}(C)$, closed under binary operation $\cdot : \mathscr{T}(C) \times \mathscr{T}(C) \to \mathscr{T}(C)$ and closed under unit $1 : \mathscr{T}(C)$. Again we define a smallest congruence relation $\cong_I$, such that:

1. it is associative, $\forall x, y, z \in \mathscr{T}(C). \quad (x \cdot y) \cdot z \cong_I x \cdot (y \cdot z)$
2. it respects identity $\forall x, y, z \in \mathscr{T}(C). \quad 1 \cdot x \cong_I x \cong_I x \cdot 1$
3. it has partial commutation
   $\forall x, z \in \mathscr{T}(C). \forall s, t \in C. \quad (s,t) \in I \Rightarrow x \cdot (s \cdot (t \cdot z)) \cong_I x \cdot (t \cdot (s \cdot z))$

Note that for the partial commutation we need to split up the term into its primitive elements, because the independence relation is not defined over terms, and only over elements. This law requires use of the unit element; since x, z can also be the unit element to denote that $s \cdot t \cong_I t \cdot s$.

Definition 7. Let C^{I*} be the *free partially commutative monoid* generated by C, that is

$$\mathscr{T}(C) / \cong_I$$

Following the same development as before: the elements of C^{I*} are the equivalence classes $[x] = \{y \in \mathscr{T}(C) \mid y \cong_I x\}$ and this structure is not a good concrete representation. To give a good concrete representation, we impose a strict total order among the elements of C. We then elect a unique representative per equivalence class. All other terms are now mapped to their unique representative. A typical representative is the lexicographic normal form. A simple algorithm for finding this normal form is as follows:

Let $s_1 \ldots s_n$ be an element of C^{I*}. For $0 < i < j \leq n$, we keep swapping two adjacent elements $s_j s_i$ into $s_i s_j$ if $(s_i, s_j) \in I$ and $s_i < s_j$ until no longer possible. The final element we obtain is the lexicographic normal form of the element we started with. Since there are only finitely many steps we can take until we reach this normal form, this is indeed a normalization procedure.

3 Formal Model

We provide an overview of the semantics of a formal model for executions of distributed algorithms. First, we characterize configurations. Given a fixed network topology and local state space and message space, configurations completely describe the global state of the modeled network and algorithm. Next, we characterize processes. By process we mean the global execution, and by node a particular executing entity. The behavior of each node is specified in terms of process descriptions. A process description (or "program") is assumed to be distributed among all nodes in the network. Every node executes the same program during the execution, resulting in the behavior of the process that was described. Process descriptions themselves are not explicitly modeled in our formal model and we assume that process descriptions are given as computable functions. Finally, we characterize the operations of the model where rules define the transitions between configurations.

3.1 State

We distinguish the following elements of a configuration: a fixed graph (V,E) that defines the topology of the network, where its vertices V are process identifiers and edges $E \subseteq V \times V$ channel identifiers; a fixed message space, denoted μ, the set of all possible messages that can be transmitted over channels; a fixed state space, denoted σ, that is a set of all of the possible local states of a single node. The channel identifier $(p,q) \in E$ is interpreted as a channel where messages are transmitted from p to q, as a single directed channel. Furthermore, we assume that for two nodes in the network that are connected there is a bi-directional unbounded FIFO channel, i.e. E is symmetric.

We model the configuration by the following functions:

Channel state label A function $\kappa : E \to \mu^*$ that maps each channel identifier to a sequence of messages in transit.

Process state label A function $\lambda : V \to \sigma$ that maps each process identifier to its corresponding local state.

A configuration is a tuple (κ, λ) that determines the global state of a network. The following functions on labeling functions are useful in the next sections.

For a process state labeling function λ, let $p \in V$ be a process identifier and $s \in \sigma$ a state. By $\lambda[p := s]$ we denote the function that replaces only the state for the given process identifier,

$$\lambda[p := s](x) \overset{\text{def}}{=} \begin{cases} s & \text{if } x = p \\ \lambda(x) & \text{otherwise} \end{cases}$$

Given a channel state labeling function κ, we define an injection function, a message predicate and a projection function, corresponding to sending and receiving, respectively. Given process identifiers $p,q \in V$ and $(p,q) \in E$ and a message $m \in \mu$. We define the injection function $\kappa[p \xrightarrow{m} q]$ that appends m to the channel between p and q,

$$\kappa[p \xrightarrow{m} q](x) \overset{\text{def}}{=} \begin{cases} \kappa(x) \mathbin{@} (m :: \varepsilon) & (p,q) = x \\ \kappa(x) & \text{otherwise} \end{cases}$$

We define the message predicate $\kappa[q \overset{m}{\leftarrow} p]?$ that holds only if message m is in the front position of the channel,

$$\kappa[q \overset{m}{\leftarrow} p]? \overset{\text{def}}{=} \exists! xs.\, \kappa(p,q) = m :: xs$$

Assume that $\kappa[q \overset{m}{\leftarrow} p]?$ holds, we then have the projection function $\kappa[q \leftarrow p]$ that constructs a new labeling function where a message is removed,

$$\kappa[q \leftarrow p](x) \overset{\text{def}}{=} \begin{cases} xs & (p,q) = x \\ \kappa(x) & \text{otherwise} \end{cases}$$

where xs refers to the unique witness of the given predicate that we assumed to hold.

We see some examples of these functions. Assume λ is a process state labeling. Given a process identifier p and two unique states $t, s \in \sigma$, then $\lambda[p := t][p := s] = \lambda[p := s]$. The last operation overwrites the previous assignment.

The channel state labeling works differently, since we have unbounded FIFO channels. For example, let κ be the empty channel labeling, i.e. $\kappa(x) = \varepsilon$ for all x. Given another process identifier q and a message $m \in \mu$, we can then construct the following channel states: $\kappa[p \overset{m}{\rightarrow} q]$ has one pending message on channel (p,q), and $\kappa[p \overset{m}{\rightarrow} q][p \overset{m}{\rightarrow} q]$ has two pending messages. Now we know that $\kappa[p \overset{m}{\rightarrow} q][q \leftarrow p] = \kappa$ since $\kappa[p \overset{m}{\rightarrow} q][q \overset{m}{\leftarrow} p]?$ holds, and the unique witness $xs = \varepsilon$. Since $\kappa[q \overset{m}{\leftarrow} p]?$ does not hold (as (p,q) is empty), the function $\kappa[q \leftarrow p]$ is undefined, thus also $\kappa[p \overset{m}{\rightarrow} q][q \leftarrow p][q \leftarrow p]$ is undefined.

3.2 Process Description

A process description is a function that maps local state to a local effect. We define local effects as follows.

Definition 8. Given a set of process identifiers V, a message space μ and a state space σ. The set of *local effects* $L(\mu, V, \sigma)$ is given by:

1. given a message $m \in \mu$, a process identifier $q \in V$ and next state $t \in \sigma$, a send effect $S(m,q,t) \in L(\mu, V, \sigma)$
2. given a function $f : \mu \times V \to \sigma$, a receive effect $R(f) \in L(\mu, V, \sigma)$
3. given a subset of states $X \subseteq \sigma$, an internal effect $I(X) \in L(\mu, V, \sigma)$

The former definition allows non-deterministic nodes by giving a choice among several next states. The internal effect without any next states, $I(\emptyset)$ is also denoted $\checkmark$, called the (local) termination effect. Furthermore, we assume that the effects are faithful, e.g. given a process identifier p that emits a send effect $S(m,q,t)$ we expect $(p,q) \in E$, and given that p receives by $f(m,q)$ then it only receives from a q such that $(q,p) \in E$.

Definition 9. A *process description* is a function $P : \sigma \to L(\mu, V, \sigma)$.

From the description of a process we can observe its behavior: each state uniquely defines a local effect. If the current state resolves to a send effect, we can update the local state and query the process description immediately again, to find out the next effect. However, if the current state resolves to a receive effect, the process becomes blocked. Its progress now depends on receiving an arbitrary message from an arbitrary node. When the current state resolves to an internal effect, we have the following two cases: 1) if the state resolved to a local termination effect, then processing at a node has ended: it can no longer send or receive any messages, 2) otherwise, we have a non-empty subset of next states, from which the next local state is a non-deterministic choice. Since in the second case we have a new local state, the process can progress by resolving the next state. The non-deterministic choice between elements of a singleton set of next states is called deterministic.

A short note on termination. We say a process does not terminate if on some node it never reaches the local termination effect. A process description that has no non-deterministic choices is called deterministic, i.e. every internal effect in its range has at most one element. Conversely, a process description with at least one such non-deterministic choice is called non-deterministic. In both cases, we have different notions of termination.

In the deterministic case, there is a possibility that a process enters an infinite loop. Consider two states s_1 and s_2. For $s_1 \mapsto I(s_2)$ and $s_2 \mapsto I(s_1)$. The local state keeps alternating between these two states indefinitely, and the process does not terminate. Similar for the non-deterministic case, where a sequence of particular non-deterministic choices can result in a similar kind of loop.

However, we also have the possibility that the progress of a process depends on a receive effect, and thus the receiving node blocks until a message arrives. The termination of such a process cannot be determined by viewing nodes in isolation, since another node must send a message for the process to make progress. We have assumed that in our model every node executes using the same process description (i.e. the same program is distributed to all nodes). The blocking of one node depends on another node with the same process description. Clearly we reach a deadlock if the process cannot make any progress in a certain configuration. Hence termination of a process description depends on deadlock freeness for every execution, under suitable fairness conditions and assumptions.

3.3 Abstract Operational Semantics

We will now define an abstract operational semantics, for evaluating a distributed algorithm: a process that executes on multiple nodes. It is abstract, and can be applied to an arbitrary process description. The modeled semantics depends on a given process descriptions. The semantics is given as a set of rules, each corresponding to a step in an execution. The semantics defines a judgment of the form $\Delta \vdash C$ where Δ is a sequence of events and C is a configuration.

We furthermore assume we are given a fixed graph (V,E), with process identifiers V and channel identifiers $E \subseteq V \times V$. We assume that we are given a message space μ and state space σ and a process description $P : \sigma \to L(\mu,V,\sigma)$. We assume that these data are fixed during the whole evaluation of the distributed algorithm.

Definition 10. The set of communication events E is given by:

1. given process identifiers $p,q \in V$ and $(p,q) \in E$ and message $m \in \mu$, a send event $(p \overset{m}{\to} q) \in E$
2. given process identifiers $p,q \in V$ and $(p,q) \in E$ and message $m \in \mu$, a receive event $(p \overset{m}{\leftarrow} q) \in E$

The sequence on the left-hand side of the judgment consists of events in E. We use a notation inspired by sequent calculus for working with the sequence, i.e. Δ, e is the (possibly empty) sequence Δ with e appended.

$$\frac{\Delta \vdash (\kappa,\lambda) \quad P(\lambda(p)) = I(X) \quad s \in X}{\Delta \vdash (\kappa, \lambda[p := s])}\ I$$

$$\frac{\Delta \vdash (\kappa,\lambda) \quad P(\lambda(p)) = S(m,q,s)}{\Delta, (p \overset{m}{\to} q) \vdash (\kappa[p \overset{m}{\to} q], \lambda[p := s])}\ S$$

$$\frac{\Delta \vdash (\kappa,\lambda) \quad P(\lambda(p)) = R(f) \quad \kappa[p \overset{m}{\leftarrow} q]? \quad f(m,q) = s}{\Delta, (p \overset{m}{\leftarrow} q) \vdash (\kappa[p \leftarrow q], \lambda[p := s])}\ R$$

Figure 2: Rules of formal model

By $P(\lambda(p))$ we denote local evaluation, i.e. given a process identifier p and its current state $\lambda(p)$ we compute a local effect by the process description. In the following rules, we universally quantify over all free variables. See Figure 3. The rules are:

- The I rule (internal event) is only applicable if some process identifier p, according to the process description P and the current local state $\lambda(p)$, results in an internal effect $I(X)$. We read X as the result of the internal effect, i.e. it is a subset of σ. If X is empty, this rule does not apply, since $s \in X$ is a (non-deterministic) choice of an element of X. We record that for the current trace Δ, it must be the case that the configuration $(\kappa, \lambda[p := s])$ holds, i.e. we update the local state space of p as the newly chosen state.
- The S rule (send event) is applicable if some process identifier p, also according to the process description P and the current local state $\lambda(p)$, results in a send effect $S(m,q,s)$ where m is the message to send, q is the recipient process identifier and s is the next state. The channel labeling is updated by including the message m, and the labeling function is updated to update the local state space of p.
- The R rule (receive event) is applicable if the process description and the current local state result in a receive effect $R(f)$. However, if there is no message to be currently received (i.e. there does not exists any q such that channel (q,p) has a message), then the process is blocked. Otherwise, we will receive the message m from neighbor q and apply the receive function f to these two data, that results in the next state for process identifier p. The channel state is updated to remove the received message, and the labeling function is updated to the local state space of p determined by f.

As stated in the first paragraph, our given operational semantics is abstract. We can instantiate this operational semantics into a concrete one by giving:

1. an algorithm specification consisting of a message space μ, a state space σ and a process description P
2. a graph (V,E) that describes the network of nodes
3. depending on the given graph, an axiom which represents the unique initial configuration of the execution

For example, an axiom could assign an initial state to each of the nodes in the given graph. Generally we expect of initial configurations that every node has the same state. However, for centralized algorithms, one special node (the initiator of the distributed algorithm) can be assigned a different state.

3.4 Partially commutative monoid

Before we can show that the sequence of events Δ is a partially commutative monoid, we need to define an independence relation between the events that we record. We first extend the rules to record more events, and then define a causal order. The complement of the causal order defines the independence relation for our former system.

Definition 11. Given a set of process identifiers V and channel identifiers $E \subseteq V \times V$, a message space μ and state space σ. The set of extends events EE is given by:

1. given process identifiers $p,q \in V$ such that $(p,q) \in E$, message $m \in \mu$ and state $s \in \sigma$, a send event $(p \xrightarrow{m} q, s) \in EE$
2. given process identifiers $p,q \in V$ such that $(p,q) \in E$, message $m \in \mu$ and state $s \in \sigma$, a receive event $(p \xleftarrow{m} q, s) \in EE$
3. given process identifier $p \in V$ and state $s \in \sigma$, an internal event $(p,s) \in EE$

As can be seen immediately, we can map extended events to communication events by dropping the extraneous state data and removing the internal events. Any independence relation defined on extended events also can be used as an independence relation on communication events. The extended rules are given in Figure 3.

$$\frac{\Delta \vdash (\kappa,\lambda) \quad P(\lambda(p)) = I(X) \quad s \in X}{\Delta,(p,s) \vdash (\kappa,\lambda[p := s])}\;I$$

$$\frac{\Delta \vdash (\kappa,\lambda) \quad P(\lambda(p)) = S(m,q,s)}{\Delta,(p \xrightarrow{m} q,s) \vdash (\kappa[p \xrightarrow{m} q],\lambda[p := s])}\;S$$

$$\frac{\Delta \vdash (\kappa,\lambda) \quad P(\lambda(p)) = R(f) \quad \kappa[p \xleftarrow{m} q]? \quad f(m,q) = s}{\Delta,(p \xleftarrow{m} q,s) \vdash (\kappa[p \leftarrow q],\lambda[p := s])}\;R$$

Figure 3: Rules of extended formal model

We make the following observation before continuing defining our independence relation. In the previous model, it is possible to determine only from the sequence of events the current channel state labeling function κ. To see how: consider the progression of the application of rules. Whenever a send event is not yet matched with a receive event, the sequence consists of more send events than receive events. Earlier send events are recorded earlier in the trace: the next message to receive at some process identified by q is the earliest unmatched send event $q \xrightarrow{m} p$ encountered. When the receive rule has been applied, a receive event is recorded and thus the send event becomes matched.

Similarly, the extended rules can be used to determine in addition the process state labeling function λ. The current local state of some process identified by p is the last event in the sequence, either a send event $(p \xrightarrow{m} q, s)$ or receive event $(p \xleftarrow{m} q, s)$ or internal event (p, s). The extended rules can thus eliminate the configuration altogether, since (κ, λ) can be determined fully based on the sequence of events Δ. However to deal with the initial configuration, we could instead require that Δ_0 is a non-empty sequence and contains precisely one internal event, for each node one. These internal events are not originated from the process description, and are regarded as pseudo-internal states only to allows us to determine (κ, λ) from Δ, also for the initial configuration.

We now look at causal order[7], which is an asymmetric relation that denotes that two events can be temporally ordered and one event happens before the other. To order these events temporally, we furthermore extend the formal model once more. Given an event $e \in EE$, we label the event by a timestamp $n \in \mathbb{N}$ that denotes its temporal position. Each application of a rule increases the timestamp by one, thereby giving each event a unique, increasing timestamp. The sequence of events Δ now consists of events $e^0, \ldots e^n$. A slight technical adjustment is needed when admitting as axiom a non-empty sequence, such that Δ starts with internal events $e^0, \ldots, e^k$ for k nodes and consists of $k+n$ events after the application of n rules.

Definition 12. A causal order is the asymmetric relation between labeled extended events such that:

1. two events $e_1^i \in EE$ and $e_2^j \in EE$, being send $(p \rightarrow \_, \_)$ or receive $(p \leftarrow \_, \_)$ or internal $(p, \_)$, with the same process identifier p and with labels $i, j \in \mathbb{N}$ are related if $i < j$
2. a send event $(\_ \xrightarrow{m} p, \_)^i$ and its corresponding receive event $(p \xleftarrow{m} \_, \_)^j$ with labels are related if $i < j$
3. it is transitive

The induced independence relation between two labeled extended events are precisely those events that are not related by the causal order in any way.

We conclude that the sequence of events forms a partially commutative monoid with this independence relation. One last remark is that this notion of “matching events” can even be captured further in a notion that is known as partially commutative inverse monoids[5]. Therein we may model x and its idempotent x^{-1} that collapse to unit $x \cdot x^{-1} = 1$, as corresponding to our earlier notion of matching send and receive events. Developing partially commutative inverse monoids is out of scope.

4 Properties and Assumptions

We categorize assumptions in the following kinds of properties. An assumption class is then a combination of one or more of these properties. We will remark on the *strength* of assumptions in comparison to other assumptions. Given two assumptions P and Q, P is a stronger assumption than Q if it is harder to satisfy the assumption P than assumption Q and furthermore Q subsumes P.

Given two assumptions P and Q. We consider that an assumption P holds if we only consider executions for which the property described by P holds. We say that P is harder to satisfy than Q if there are less executions when P holds than when Q holds.

For example, take "network of size 3" and "network with 6 edges". We argue that "network with 6 edges" has more executions; there are more networks with 6 edges than there are with 3 vertices, hence "network of size 3" is stronger.

Subsumption will be defined in terms of algorithm correctness. Correctness is informally defined as provability of certain properties that hold for every execution. For example, the statement "algorithm x terminates" is a correctness property and states that for every execution, every node eventually locally terminates. Given a correctness property ϕ, and assumptions P and Q, we define that Q subsumes P if and only if, assuming ϕ holds under Q then ϕ necessarily holds under P.

Take as example the assumptions "algorithm x computes correct answer" and "algorithm x terminates". An algorithm has a correct answer, if all states of nodes before local termination are consistent according to some desired property, e.g. the algorithm has computed a sink tree. An algorithm that terminates may terminate without a correct answer. The former subsumes the latter since an algorithm must terminate before computing a correct answer.

The moral is: stronger assumptions lead to simpler algorithms, but stronger assumptions are less applicable to the real world. In our development of the formal model in previous sections, we have implicitly introduced a few assumptions. We now discuss the possible extension of the formal model along different choices of assumptions.

4.1 Showing properties

The formal system outlined in previous section can be used to show properties of distributed algorithms. Given a distributed algorithm, we need to encode it as a process description. Then we instantiate the rules using the given process description. The resulting rules can be interpreted as rewrite rules. It is strongly suspected that termination of the distributed algorithm corresponds with the termination of the rewrite system. Furthermore, as we have seen that the configuration is determined by the sequence of events; it is foreseeable to have judgments of the form $\Delta \vDash \phi$, where ϕ is a property.

A full development of a logical specification system on top of our given semantics is out of scope for the current research. However, it is expected that a usual notion of truth can be defined in terms of evaluating ϕ based on a configuration, e.g. allowing the expression of modal properties such as "eventually the elementary state of every node is a fully-developed sink tree towards the initiator," which is a useful property for Cidon's algorithm in Section 5.

4.2 Channel assumptions

A channel models the communication between two nodes. In the following discussion we abstract over the message contents, i.e. arbitrary messages of arbitrary size are considered as one atomic unit. Operations of channels are sending and receiving an arbitrary message. We distinguish the following kinds of channels:

Lossless channels Every message is transmitted as-is: every send operation corresponds to a receive operation.

Duplicating channels A message could be duplicated while it is in transit: every send corresponds to one or more receiving operations.

Dropping channels A message could be dropped while in transit: every send corresponds to zero or one receiving operations.

Unreliable channels A message could both be dropped or duplicated: every send corresponds to zero or more receiving operations.

We have previously modeled only lossless channels: every send corresponds to a receive. To model duplicating channels, we need to introduce another rule.

$$\frac{\Delta \vdash (\kappa, \lambda) \quad \kappa[q \overset{m}{\leftarrow} p]?}{\Delta \vdash (\kappa[m :: (p,q)], \lambda)} \; Du$$

where $\kappa[m :: (p,q)](x) = \begin{cases} m :: \kappa(x) & \text{if } (p,q) = x \\ \kappa(x) & \text{otherwise} \end{cases}$.

This rule allows one to duplicate a message if it is in front of the buffer. From this description it follows immediately that nodes might become overwhelmed by an infinite number of the same message.

Modeling dropping channels amounts to the following rule.

$$\frac{\Delta \vdash (\kappa, \lambda) \quad \kappa[p \overset{m}{\leftarrow} q]?}{\Delta \vdash (\kappa[p \leftarrow q], \lambda)} \; Dr$$

Compare it with the rule R for receiving messages: this rule drops the message but does not take account of the local effect. A node that blocks on receiving a message might never make any progress due to this rule. As with duplicating channels overwhelming a node, this rule allows us to starve the node. It is an open question to come up with a useful notion of fairness, to prevent these extreme situations. Admitting both rules allows one to model unreliable channels.

Another aspect of a channel is the number of messages in transit:

Bounded buffers The maximal number of messages in transit is bounded by some fixed natural number $n \geq 1$.

Unbounded buffers The number of messages in transit is not bounded.

Modeling these buffers amounts to changing the channel state labeling function. For unbounded buffers we have $\kappa : E \to \mu^*$, but for bounded buffers, we need $\kappa : E \to \bigcup_{i=0}^{n} \mu^i$, i.e. lists of at most length n. Bounded buffers are related to unbounded dropping channels. Suppose we have a bounded buffer that is full. If we send a message through the full channel it gets dropped. This can be simulated by an unbounded buffer by dropping the same message just before one would normally receive it.

For channels which have more than one message in transit, we may not or may allow the reordering of corresponding operations:

FIFO channels The order of send operations is related to the order of corresponding receive operations, as a FIFO queue.

Commutative channels The order of send operations is unrelated to the order of corresponding receive operations.

The order is normally preserved by the list structure of the channel state labeling function; if we instead map channel identifiers to multisets $\mathbb{N}^{\mu}$, the relative order of messages is no longer significant. This nicely composes with the other notions; e.g. duplicating commutative channels allow for messages to be duplicated and then commuted. The duplicated message may be interleaved with other unrelated messages, a late duplicated arrival. In case of the bounded variant, we can limit the total number of elements of the multiset: sending a message through a full channel then also drops the newly sent message.

4.3 Node assumptions

Nodes receive information from their environment, and locally operate on received information. In anonymous networks, nodes have no information regarding their own identity. We distinguish the following properties:

Anonymous nodes The node has no a priori information available to uniquely identify itself among other nodes in the network.

Uniquely identified nodes Each node has a unique identifier that distinguishes it from all other nodes in the network.

Our former model assumes uniquely identified nodes; every neighboring node knows from which node in the network it has received a message. Anonymous nodes can only identify the different channels to neighbors, but not necessarily know how these relate this information to a global process identifier. For example, the function $f : \mu \times V \to \sigma$ of the receive effect can be adapted to instead range of $\mu \times \mathbb{N} \to \sigma$, where for each node its neighbors get assigned an index as natural number. If we have two different nodes, with possibly different but the same number of neighbors, they both get identified by the same natural numbers counting from zero upward. Also, the send effect must be adapted to instead require the index of the neighbor to send a message to, instead of the global process identifier. Each send effect is now interpreted within the context of the sending node, since two nodes can potentially share the same indexing of its neighbors but identify different nodes.

Crashing nodes and Byzantine nodes could potentially be modeled by admitting special internal events (to model perfect failure detectors) and admitting additional rules (for injecting arbitrary messages from nodes that have a Byzantine failure), but are out of the scope of the current research.

4.4 Topology assumptions

A network topology is typically classified by distinguishing the following properties:

Directed An edge between two nodes need not be symmetric.

Undirected An edge between two nodes necessarily is symmetric.

Furthermore, we distinguish the following connection properties, that is, reachability between arbitrary nodes:

Fully connected Each node has an edge to every other node in the network.

Strongly connected Each node is reachable from every other node in the network.

Weakly connected Given two nodes p and q, either p is reachable from q, or q is reachable from p.

Not connected There exists two nodes p and q, such that neither p is reachable from q nor q is reachable from p.

Clearly strongly connected and weakly connected topologies coincide when edges are undirected.

The network topology has no influence on our modeling; it is only a property of the fixed but arbitrary network (V,E). However, we do not deal with the possibility of faulty process descriptions, e.g. a process description that sends a message over a non-existing channel. In the case of dynamic networks, i.e. networks with changing topology during the execution of an algorithm, it is necessary to deal with such faulty descriptions. Furthermore, to model crashing nodes it is first necessary to model dynamic networks, e.g. a crashing node corresponds by the disappearance of all channels of such a crashing node.

5 Cidon's Algorithm

We study Cidon's distributed depth-first algorithm[3]. It was found that the correctness proof in the original paper is not correct. Also, the original paper mentions a worst-case message complexity of $3|E|$, while others[9, 10] mention a worst-case message complexity of $4|E|$. We present the algorithm by modeling it in our framework: by giving a message space, state space and process description.

Definition 13. Message space μ for Cidon's algorithm is the set $\mu \overset{\text{def}}{=} \{\text{Token}, \text{Info}\}$, viz. a token message and an information message.

The following definition is a typical inductive data structure: the four elements are called the constructors. The elementary state space is called *elementary* because it contains the essential information after an execution terminates. We can read the result of Cidon's algorithm from this part of the state space alone. We call p the parent of the node that has either elementary state $\text{Received}(p)$ or $\text{Replied}(p)$.

Definition 14. The elementary state space σ^{el} of Cidon's algorithm, given a channel identifier $p \in V$, is either: $\text{Received}(p) \in \sigma^{el}$, $\text{Replied}(p) \in \sigma^{el}$, $\text{Undefined} \in \sigma^{el}$ or $\text{Initiator} \in \sigma^{el}$.

Definition 15. The state space σ of Cidon's algorithm is the tuple $(t, s^{el}, n^P, n^F, n^I) \in \{0,1\} \times \sigma^{el} \times (V \uplus \{u\}) \times \mathscr{P}(V) \times \mathscr{P}(V)$, where

1. $t \in \{0,1\}$ is a Boolean that indicates whether the node currently has the token
2. $s^{el} \in \sigma^{el}$ is the elementary state, indicating what part of the algorithm is currently executing and is used after termination to read off results

3. $n^P \in V \uplus \{u\}$ is the (potential) promised neighbor to forward the token to, and is used to keep track of neighbors to send information messages to, either having some process identifier $n^P \in V$ or the (fixed) undefined value $n^P = u$
4. $n^F \subseteq V$ is a set of forwarding neighbors, that initially consists of all neighbors for any given node
5. $n^I \subseteq V$ is a set of information neighbors, the neighbors to which no information message has been sent yet

The following definition gives the gist of the algorithm. We can read the following equations as pattern matching on the left-hand side, as is usual in functional programming languages. All free variables are quantified over universally, unless stated otherwise. Remark the occurrence of the parent p, often as part of the elementary state $\text{Received}(p)$ or $\text{Replied}(p)$: this variable is matched by destructing the value of s^{el}.

Definition 16. Let $P : \sigma \to L(\mu, V, \sigma)$ be the process description of Cidon's algorithm, defined as follows:

$$P(1, s^{el}, u, n^F, n^I) = I(\{(1, s^{el}, n^P, n^F \backslash n^P, n^I \backslash n^P) \mid n^P \in n^F\}) \tag{1}$$
$$\text{where } s^{el} \in \{\text{Initiator}, \text{Received}(p)\} \text{ and } n^F \neq \emptyset$$

The node has the token, and must promise an intended neighbor to forward the token to. The choice of which neighbor is chosen is non-deterministic. After choosing a neighbor, the node no longer forwards a token or sends an information message to the chosen neighbor. This is the only internal non-deterministic rule of the algorithm.

$$P(1, s^{el}, n^P, n^F, n^I) = S(\text{Info}, i, (1, s^{el}, n^P, n^F, n^I \backslash i)) \tag{2}$$
$$\text{where } s^{el} \in \{\text{Initiator}, \text{Received}(p)\} \text{ and } n^P \neq u \text{ and } i \in n^I$$

A promised neighbor is selected and the node sends information messages to all its other neighbors. Repeated as long as the neighbors to send an information message to is non-empty. The order of choosing neighbors is already non-deterministic by non-determinism of arrival of messages.

$$P(1, s^{el}, n^P, n^F, \emptyset) = S(\text{Token}, n^P, (0, s^{el}, n^P, n^F, \emptyset)) \tag{3}$$
$$\text{where } s^{el} \in \{\text{Initiator}, \text{Received}(p)\} \text{ and } n^P \neq u$$

Once all neighbors are informed, the node proceeds to send the token to its promised neighbor. It still references this neighbor in case it receives a message back (see 9).

$$P(1, \text{Received}(p), u, \emptyset, n^I) = S(\text{Info}, i, (1, \text{Received}(p), u, \emptyset, n^I \backslash i)) \tag{4}$$
$$\text{where } i \in n^I$$

The node has a token but no forwarding candidates left. It must still make sure that it has informed all its neighbors of its possession of the token. Eventually, the set of neighbors to inform will be empty. Since the promised neighbor is undefined, this rule only applies to only a few cases: consider that a node has received Info messages from all its neighbors, and finally receives the Token. It cannot choose a promised neighbor, because there is none. It must still inform its neighbors: afterwards return the Token back to its parent.

$$P(1,\text{Received}(p),u,\emptyset,\emptyset) = S(\text{Token},p,(0,\text{Replied}(p),u,\emptyset,\emptyset)) \tag{5}$$

The node has the token, no forwarding candidates left and no neighbors to inform. The token is sent back to the original neighbor, called its *parent*, from which the node received the token beforehand. A peculiar case is in a dead-end, where the token is immediately returned back to its parent.

$$P(1,\text{Initiator},u,\emptyset,\emptyset) = \checkmark \tag{6}$$

If the initiator has the token, no candidates left and no neighbors to inform (as in rule 5), it locally terminates.

$$P(0,\text{Replied}(p),u,\emptyset,\emptyset) = \checkmark \text{ for any } p \in V \tag{7}$$

Under similar conditions, if a node has already returned the token back to its parent, it locally terminates.

$$\begin{gathered} P(0,\text{Undefined},u,n^F,n^I) = R(f) \\ f(\text{Token},p) = (1,\text{Received}(p),u,n^F\backslash p,n^I\backslash p) \\ f(\text{Info},p) = (0,\text{Undefined},u,n^F\backslash p,n^I) \end{gathered} \tag{8}$$

A node which has no token, is a non-initiator and has no parent, waits for receiving a message. Depending on the kind of message, it either registers the message sender as its parent in case of a token (and removes the parent from its neighbors), or in case of an information message removes the sender as a candidate.

$$\begin{gathered} P(0,s^{el},n^P,n^F,n^I) = R(f) \\ \text{where } s^{el} \in \{\text{Received}(p),\text{Initiator}\} \text{ and } n^P \neq u \\ \text{and } \begin{array}{l} f(m,x) = (1,s^{el},u,n^F,n^I) \text{ where } x = n^P \\ f(m,q) = (0,s^{el},u,n^F\backslash q,n^I) \text{ for any } q \in V \text{ s.t. } q \neq n^P \end{array} \end{gathered} \tag{9}$$

A node which has no token, but still keeps track of a promised neighbor (see step 3), must wait for any kind of message. If the message is from its promised neighbor, it is interpreted as returning back the token (either because the token is literally given back, or because of a late information message). Otherwise, the message came from a different neighbor, in which case it is simply removed as a candidate (as in step 10).

$$\begin{gathered} P(0,s^{el},u,n^F,n^I) = R(f) \\ \text{where } s^{el} \in \{\text{Received}(p),\text{Initiator}\} \\ \text{and } \begin{array}{r} f(\text{Token},q) = (1,s^{el},u,n^F,n^I) \\ f(\text{Info},q) = (0,s^{el},u,n^F\backslash q,n^I) \end{array} \end{gathered} \tag{10}$$

In any other case, if a node has no token and not sent a token out, it either accepts a sent token and handles it as given, or if receiving an information message it removes the sending neighbor.

This ends the definition of the algorithm. Before we can instantiate the abstract operational semantics and obtain an operation semantics that models the execution of this algorithm, we need to give an initial configuration. We let the initial configuration depend on a (fixed) graph (V,E). Let $E(p) = \{q \mid (p,q) \in E\}$ denote the outgoing neighbor set of p. Let κ be the empty channel state labeling, i.e. $\kappa(x) = \varepsilon$. Pick any $i \in V$ to be the initiator. Let $\lambda(i) = (1, \text{Initiator}, u, E(i), E(i))$ and for any $x \neq i$ let $\lambda(x) = (0, \text{Undefined}, u, E(x), E(x))$.

We can ask ourselves the following critical questions regarding this encoding of Cidon's distributed depth-first search algorithm:

1. Why is the given encoding precisely Cidon's distributed depth-first search algorithm? Without regression into philosophical identity questions, we can pragmatically state that an algorithm is correctly realized if every property of the ideal algorithm can also be observed from its implementation.
2. What is the expected result of executing this algorithm? Once we choose a graph and an initiator, we can perform the unraveling of the operational semantics and obtain any execution. Expected properties are: every execution terminates, i.e. every node is locally terminated, and afterwards we observe that the elementary state represents a sink tree towards the initiator. Every node has as elementary state either $\text{Replied}(q)$ for some $q \in V$ or is Initiator (see 6 and 7). The algorithm is correct if for every non-initiator, the path one obtains by following the argument to Replied is finite and ends with i, the initiator as given by the initial configuration.
3. Additionally, one can expect the following property: the Boolean value t in the state of each process is only 1 if the node "possesses" the token. This means that for every non-initiator: between a receive event of a Token and the next send event of the Token, this part of the state must be 1 and 0 otherwise. There is one exception to this property, namely that a node that sends a Token and receives instead an Info message back from its promised neighbor n^P, it again possesses the token (see 9). We could adapt our definition to explicitly return the Token in case of wrongfully receiving it; however, at the cost of an unneeded message. This trade-off shows that encoding an algorithm is a matter of design: choosing between elegance of definition and practical efficiency.

The algorithm as it is presented here is, with slight modifications, also implemented in the DaViz tool mentioned in the introduction. It is possible for interested readers to explore executions of Cidon's distributed depth-first search on user-defined graphs: DaViz shows each event in a timeline and allows the user to inspect the state space assigned to any node. In addition, messages in transit can be inspected. The tool also makes a nice illustration of partially commutative monoids as represented as directed acyclic graphs in its timeline window, and allows users to change the linearization of traces.

6 Conclusions

Over the course of this paper, we have developed two notions. The first is the development of monoids, possibly freely generated, and its many variants: the most important one being the partially commutative monoid. We have also seen a formal model, which is given by a few rules that can be unraveled by supplying so-called process descriptions (or "programs"). We have explored a few assumptions and related them to the formal model, and shown some dependencies between assumptions. Finally, we have seen a concrete example of a process description that realizes Cidon's distributed depth first search algorithm.

The introduction mentions that this research was executed with two concerns in mind:

1. Do the underlying structures reveal useful decompositions of algorithms?
2. What is necessary of a formal model by which properties of distributed algorithms can be proven correct?

We can finally give a (partial) answer to these two questions as follows:

1. Algorithms can be encoded as process descriptions. Process descriptions can themselves be composed by combining other process descriptions. It might be useful to explore these compositions, and develop an expression language in which algorithms can be encoded more easily, for the following reasons: (a.) the presented model allows one to evaluate and simulate given process descriptions, i.e. they are actionable, (b.) as can be seen from Cidon's example, it might require much effort to work directly with the low-level encoding, and thus a high-level approach of composing algorithms as modules seems fruitful.
2. We have seen a suggestion which pointed into the direction of proving termination of process descriptions, before one can use them as a structure to prove properties. It might be useful to explore the development of a framework in which: (a.) specifications of configurations can be expressed in a (modal) logic, (b.) research the possibility of a syntactic system by which the same properties can be derived. These two combined might lead to what the author envisions as a "certified distributed algorithms" toolkit.

Moreover, during the research parts of the results are formalized in Coq. This exercise was helpful for structuring this report; the level of detail required for formalizing monoids in Coq helped the author to understand the literature more thoroughly. Not every aspect detailed in this paper was formalized due to two reasons: (a.) for studying most aspects related to the formal model, a detailed formalization can only be made after the precise definitions are written down. It is easier to change such definitions while they are not yet set in stone: changing the definition mostly invalidates any lemma proven thereafter, requiring much of the rework of the development. (b.) formalizing in Coq is very time intensive, and may or may not have the expected results. Failure to prove a lemma in Coq does not imply that the lemma is not provable; intuitively clear notions are hard to work with in a formalized setting, and finding useful representations to work with these notions is hard.

References

[1] Baruch Awerbuch. A new distributed Depth-First-Search algorithm. *Information Processing Letters*, 20(3):147–150, 1985.

[2] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq Proof Assistant Reference Manual: Version 6.1. Technical Report 0203, INRIA, 1997.

[3] Israel Cidon. Yet another distributed depth-first-search algorithm. *Information Processing Letters*, 26(6):301–305, 1988.

[4] Cyril Cohen. Pragmatic Quotient Types in Coq. In *International Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 213–228. Springer, 2013.

[5] Volker Diekert, Markus Lohrey, and Alexander Miller. Partially commutative inverse monoids. *Semigroup Forum*, 77(2):196–226, 2008.

[6] Volker Diekert and Yves Métivier. Partial commutation and traces. In *Handbook of formal languages*, pages 457–533. Springer, 1997.

[7] Wan Fokkink. *Distributed Algorithms: An Intuitive Approach.* MIT Press, 2013.

[8] Dominique Perrin. Words over a Partially Commutative Alphabet. In *Combinatorial algorithms on words*, volume 12 of *NATO ASI*, pages 329–340. Springer, 1985.

[9] Mohan B. Sharma, Narasimha K. Mandyam, and Sitharama S. Iyangar. An optimal distributed depth-first-search algorithm. In *Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 287–294. ACM, 1989.

[10] Yung H. Tsin. Some remarks on distributed depth-first search. *Information Processing Letters*, 82(4):173–178, 2002.