Facilitating the Verification of Diffusing Computations and Their Applications

Tanja E.J. Vos
Instituto Tecnológico de Informática
Universidad Politécnica de Valencia, Camino de Vera s/n, 46071 Valencia, Spain
tanja@iti.upv.es

Abstract

We study a class of distributed algorithms, generally known by the name of diffusing computations, that play an important role in all kinds distributed and/or database applications to perform tasks like termination detection, leader election, or propagation of information with feedback. We construct a highly parameterized abstract algorithm and shown that many existing algorithms and their applications can be obtained from this abstract algorithm by instantiating the parameters appropriately and/or refining some of its actions. Subsequently, we show that this use of parameterization and re-usability of notation and proof leads to a reduction of the effort and cost of developing and verifying distributed diffusing computations. More specific, we show that proving the correctness of any application now boils down to verifying an application-specific safety property and reusing the termination and safety proofs of the underlying abstract algorithm.

Keywords: Parameterization, re-use, specifications, formal proof, distributed algorithms, diffusing computations.

1 Introduction

It is our belief that the use of parameterization en re-usability of notation and proof are not sufficiently used when specifying and proving distributed algorithms. This paper uses a family of diffusing computations to show how easily these concepts can be used during the specification of distributed algorithms. We will show how this can lead to better representations of distributed algorithms that: show similarities with (and differences from) related algorithms, increase pedagogical effectiveness, and reduce proof-effort of specific applications of these algorithms. The concepts used in this paper are not new, nor are the algorithms presented. However, specifying a class of distributed algorithms using parameterization and re-usability of notation and showing the many advantages stated above, has, in our opinion, not been sufficiently explored.

2 Diffusing Computations

Diffusing computations, or DCs, were first introduced in [11]. They describe the class of distributed algorithms that first generate a rooted spanning tree (RST) in a connected network of nodes, and then use this tree in order to achieve some goal. Diffusing computations play an important role in computer science since they can be used for: *propagation of information with feedback* [25]; finding the ordering of all node identities in a connected network of nodes [6], and based on this performing *leader election* [29]; computing shortest paths and based on that perform *routing* [2, 15]; *global termination detection* [11, 6, 3, 14, 20, 21, 28, 29]; *deadlock detection* [4, 19] *(re-)synchronization* [12, 24]; *distributed reset* [1]; computing a function of which each node holds part of the input (e.g. summation, maximum finding, distributed approximation of network diameter) [21, 29] and many other *pyramidal functions* [22].

DCs occur under different names in different papers: distributed termination algorithm [14], wave algorithms [29], and total algorithms [28]. Specific instantiations and variations of diffusing computations are TARRY's algorithm, a traversal algorithm for arbitrary networks given by Tarry [27]; ECHO algorithm, introduced by Chang [6]; the classical Depth First Search algorithm (DFS) [7, 29], Segall's PIF [25] algorithms, and the Dijkstra-Scholten Algorithm (DS) [11, 29].

Diffusing computations constitute an interesting case study since they are highly parallel and nondeterministic. Consequently, many have studied them. Less formal work includes [25, 26, 28, 29]. More formal work can be found in [6, 2, 20, 15, 8, 9, 18, 17, 30, 13, 10]. Examining and studying all these works we concluded that specifications and correctness proofs are most of the time not easy to reuse in order to demonstrate the various applications of diffusing computations mentioned above.

3 Terminology and Notation

In this paper we use UNITY because it is simple to explain in a short paper like this. We could, however, have used any action-based formalism. Thus, we do not advocate that UNITY leads to better representations. Each formalism has its advantages and disadvantages, this paper does not go into these details.

UNITY [5] programs Π consist of a predicate (**ini**Π) specifying the initial condition of the program, a set of variable declarations (**v**Π), and a set of actions (**a**Π) separated by the symbol $[\!]$. Actions in a UNITY program (elements of the set Action) can be (multiple) assignments or guarded (if-then-else) actions. Simultaneous execution of assignments is modeled by $\|$, e.g. the action $x := 1 \parallel y := 2$ has the same effect as the multiple assignment $x, y := 1, 2$. A program execution is infinite, in each step an action is selected nondeterministically and executed. Selection is weakly fair, i.e. every action is selected infinitely often.

Program states will be modeled by functions from variables (`Var`) to values (`Val`). State-predicates (elements of Pred) are sets of states modeled by functions from states to bool. State-expressions (elements of Expr) are functions from states to `Val`. When it is clear from the context, operators will be overloaded to denote their state-lifted counterparts.

A state-predicate J is *stable* in Π (denoted ${}_{\Pi}\vdash \circlearrowleft J$), if once J holds during the execution of Π, it will remain to hold. A stable predicate is an *invariant* when it also holds in the initial state of the program.

We will use the original UNITY [5] operators unless (to specify safety properties) and ensures (to specify one-step progress properties). To denote general progress properties, we will use Prasetya's *reach* ($\rightarrowtail$) operator [23]. The most important reason for this choice was the fact that Prasetya has embedded this UNITY variant in the HOL theorem prover [16, 23]. Using this HOL library with its wealth of pre-proved theorems and inference-rules, we saved a lot of time proving the diffusing computations. Details about these mechanical verification activities are outside the scope of this paper and can be found in [32].

Properties involving Prasetya's *reach* operator look like this: $J \ {}_{P}\vdash\ p \rightarrowtail q$ and, intuitively, mean that if J is stable in program P, then P can make progress from $J \wedge p$ to q. Like leadsto, the *reach* operator is defined as the smallest transitive and disjunctive closure of ensures. However, *reach* requires that the state-predicates

p and q are restricted to be confined by the write variables of program P, which means that $p \rightarrowtail q$ describes progress made through the writable part of a program.

Prasetya [23] also introduced a convergence operator denoted by $J \,{}_{\Pi}\!\vdash p \rightsquigarrow q$. Intuitively, this means that a program Π *converges* from p to q under the stability of J, if, given that ${}_{\Pi}\!\vdash\!\circlearrowleft J$, program Π started in state-predicate p will eventually find itself in a situation where q holds and remains to hold. Laws about $\rightarrowtail$ and $\rightsquigarrow$, needed in this paper, are listed in Appendix 11 and taken from [23].

Function application is represented by a dot, e.g. $s.x$ applies function s to x. Function composition is denoted by $\circ$. In proofs we will, when convenient, replace $\wedge$ by a newline. For example: $p \Leftarrow$ (some theorems and definitions) $q \wedge r$, will be written like:

p
$\Leftarrow$ (some theorems and definitions)
q
r

4 The Communication Network

The communication networks are connected *centralized networks* employing *bi-directional asynchronous communication*. These networks are modeled by a triple ($\mathbb{P}$, neighs, *starter*), where: $\mathbb{P}$ is a finite set of nodes; neighs is a function that given some node $p \in \mathbb{P}$, returns the set of neighbors of p, i.e. the set of nodes that are connected to p by a bi-directional communication link; and *starter* is a node in $\mathbb{P}$ that distinguishes itself from all other nodes (called the *followers*), in that it can spontaneously start the execution of its local algorithm (e.g. because it is triggered by some internal event). The *followers* can only start execution of their local algorithm after they have received a message from some neighbor. Such a network is *connected* if every pair of nodes is connected by a path of communication links.

For this paper it is sufficient to give an abstract model of *asynchronous communication*. We do this by just stating the functionality of the communication primitives. These primitives, listed below, only assign to a specially designated set of Communication Variables, denoted by CV.($\mathbb{P}$, neighs, *starter*):

- mit.$p.q$ (acronym for message in transit) can be used to check for a message in transit from p to q;
- p nr_sent_to q, enables nodes to check how many messages they have already sent to a neighbor q;
- p nr_rec_from q, to check the amount of messages received from q.
- send.$p.q.m$, implements that a node p sends message m to q;
- The receive action contains two subtleties. First, it only has the desired effect when a message is indeed in transit. So the programmer has to ensure that this action is only executed after checking and confirming the availability of a message to receive. Second, when a node p receives a message m from say q, p usually does something with the received value(s) and stores the result(s) somewhere. Since we have no sequential composition of actions in UNITY, receive is parameterized with a function that when applied to p, q and m, returns an assignment that processes the received message and assigns the results to the appropriate variables. So receive.$p.q.a$ implements that p receives a message m from q and processes it using assignment $a.p.q.m$.

Any program that wants to use these communication primitives should incorporate ASYNC_Init.($\mathbb{P}$, neighs, *starter*) in its initial condition in order to initialize the communication variables.

5 Analyzing Diffusing Computations

Most of the diffusing computation algorithms found in literature more or less have the following behavior. A *follower* becomes active when it receives its very first message, and it marks the node from which it received this first message as its *father*. A non-*idle* node proceeds with two activities: *propagation*, i.e. sending a message to all its neighbors except its father; and *collecting* one message from each of its neighbors. When the *starter* has sent and received one message to and from all its neighbors (i.e. has completed *propagating* and *collecting*), it immediately is *done*, whereas a *follower* node p has completed *propagating* and *collecting* it first has to report to its father prior to becoming *done*.

The differences between the algorithms are in the communication protocols, more specifically when non-*idle* nodes are allowed to collect or propagate a message:

prog Π

init ASYNC_Init.($\mathbb{P}$, neighs, *starter*) $\wedge$ $init_\Pi$

var CV.($\mathbb{P}$, neighs, *starter*) $\cup$ $\{p \in \mathbb{P} \mid \mathsf{idle.p}\}$ $\cup$ $\{p \in \mathbb{P} \mid \mathsf{father.p}\}$

assign

$[\!]_{p \in \mathbb{P}}$

$[\!]_{q \in \mathsf{neighs}.p}$ **if** idle.p $\wedge$ mit.$q.p$

then receive.$p.q.\langle$**process**$\rangle$ $\|$ father.$p := q$ $\|$ idle.$p :=$ false (IDLE)

$[\!]$

$[\!]_{q \in \mathsf{neighs}.p}$ **if** $\neg$ idle.p $\wedge$ mit.$q.p$ $\wedge$ $collecting_\Pi.p.q$

then receive.$p.q.\langle$**process**$\rangle$ (COL)

$[\!]$

$[\!]_{q \in \mathsf{neighs}.p}$ **if** $\neg$ idle.p $\wedge$ *can_propagate.p.q* $\wedge$ $propagating_\Pi.p.q$

then send.$p.q.\langle$**mes**$\rangle$ (PROP)

$[\!]$

if *finished_col_and_prop.p* $\wedge$ $\neg$*reported_to_father.p*

then send.$p.$(father.p).$\langle$**mes**$\rangle$ (DONE)

Figure 1: Skeleton of diffusing computation Π.

- The DS algorithm allows a node to freely merge its propagating and collecting actions as long as it has not yet received messages from all its neighbors, and it has not yet sent to all its neighbors that are not its father.
- In the ECHO and PIF algorithms, a non-*idle* node p can only receive a message after p has sent messages to all its neighbors except its father. So, the *propagating* activities must be completed before *collecting* information from non-father-neighbors.
- In the TARRY algorithm, a non-*idle* node p can only propagate to a neighbor if the last event of p was a receive event; otherwise it has to wait until it receives something. So, the *propagating* and *collecting* activities strictly alternate.
- In the DFS algorithm, a non-idle node p in its propagating phase whose last event was receiving a message from some neighbor q: **if** p can propagate a message back to q, i.e. q is not p's father, and p has not yet sent to q, **then** p has to send a message back to this node q **otherwise** it can act like in TARRY, and just pick any non-father-neighbor to which it has not yet sent a message.

5.1 Construct the Abstract Algorithm

Based on the described similarities, we can construct a first skeleton for all local algorithms (including the one for the *starter*). The skeleton is shown in Figure 1. The differences are clearly indicated by sub-scripting the program guards and part of the initial condition of which we have not yet given the exact characterization. Furthermore, the skeleton abstracts from specific applications of the algorithms by leaving the contents of the messages that are being sent, and the ways these are processed upon receipt, unspecified. These two aspects do not only increase the re-usability of the specification, it also gives the reader the opportunity to develop his or her own feeling about the possible uses of the algorithm. This improves pedagogical effectiveness because it can help the reader to better understand the existing applications, and can result in ingenious new ones.

5.1.1 Capturing the Similarities

Besides the similarities revealed by the skeleton in Figure 1, the analysis also taught us that: when a node is *collecting* this implies that it has not yet received messages from all its neighbors**;** when a node is *propagating* this implies that it has not yet sent to all its neighbors that are not its father**;** *p can propagate to q* when p has not yet sent to q, and q is not its father**;** when a node is *finished_col_and_prop*, then it has received from all its neighbors and it has sent to all its non-father-neighbors**;** when a node has not yet reported to its father, it has not yet sent a message to its father **;** when a node p is *done* it has sent and received a message to and from all of its neighbors (i.e. including its father). To capture these similarities we define the following predicates:

$rec\_from\_all\_neighs.p = \forall q \in \mathsf{neighs}.p : p\ \mathsf{nr\_rec\_from}\ q = 1$

$can\_propagate.p.q = \ p\ \mathsf{nr\_sent\_to}\ q = 0\ \wedge\ q \neq \mathsf{father}.p$

$reported\_to\_father.p = \quad (p\ \mathsf{nr\_sent\_to}\ (\mathsf{father}.p) = 1)\ \vee\ (p = starter)$

$sent\_to\_all\_neighs.p = \forall q \in \mathsf{neighs}.p\ : p\ \mathsf{nr\_sent\_to}\ q = 1$

$sent\_to\_all\_non\_fathers.p = \forall q \in \mathsf{neighs}.p : q \neq \mathsf{father}.p\ \Rightarrow\ p\ \mathsf{nr\_sent\_to}\ q = 1$

$finished\_col\_and\_prop.p = rec\_from\_all\_neighs.p\ \wedge\ sent\_to\_all\_non\_fathers.p$

$done.p = rec\_from\_all\_neighs.p \wedge sent\_to\_all\_neighs.p$

5.1.2 Handling the Differences

Differences occur when non-*idle* nodes are allowed to collect or propagate a message, i.e. in the characterization of the *collecting* and *propagating*. In the ECHO and PIF algorithm, the *propagating* activities must be completed before a node can start *collecting* from non-father-neighbors. Consequently:

$propagating_{\text{ECHO}}.p.q = \neg\ sent\_to\_all\_non\_fathers.p$

$collecting_{\text{ECHO}}.p.q = \neg\ rec\_from\_all\_neighs.p\ \wedge\ \neg propagating_{\text{ECHO}}.p.q$

$init_{\text{ECHO}} = \mathsf{father}.starter = starter \wedge \forall p \in \mathbb{P} : p = starter \neq \mathsf{idle}.p$

Instantiating Π with ECHO in Figure 1 specifies the whole ECHO algorithm! The DS algorithm allows a node to freely merge its propagating and collecting actions. Consequently, the *propagating* and *collecting* predicates for this algorithm are:

$propagating_{\text{DS}}.p.q = \neg\ sent\_to\_all\_non\_fathers.p$

$collecting_{\text{DS}}.p.q = \neg\ rec\_from\_all\_neighs.p$

$init_{\text{DS}} = init_{\text{ECHO}}$

Substituting Π for DS in Figure 1 constitutes the DS algorithm.

In the TARRY algorithm, a non-*idle* node p can only propagate to a neighbor if the last event of p was a receive event. We can represent this by introducing a new boolean-typed variable $\mathsf{le\_rec}.p$ (i.e. last event was a receive) for every node p, which we initialize to false for all nodes except the *starter*. In order to give the variable the right value, i.e. whether the last event of p was a receive event, we can add the assignment ($\mathsf{le\_rec}.p := \mathsf{true}$) to the **then** clauses of the actions {IDLE, COL}, and ($\mathsf{le\_rec}.p := \mathsf{false}$) to the **then** clauses of {PROP, DONE}. Finally, we can characterize *collecting* and *propagating* as follows:

$propagating_{\text{TARRY}}.p.q = \neg\ sent\_to\_all\_non\_fathers.p\ \wedge\ (\mathsf{le\_rec}.p)$

$collecting_{\text{TARRY}}.p.q = \neg\ rec\_from\_all\_neighs.p\ \wedge\ \neg(\mathsf{le\_rec}.p)$

$init_{\text{TARRY}} = init_{\text{ECHO}}\ \wedge\ \forall p \in \mathbb{P} : (p = starter) \neq (\neg \mathsf{le\_rec}.p)$

prog Π

init ASYNC_Init.($\mathbb{P}$, neighs, *starter*) $\wedge$ $init_\Pi$

var CV.($\mathbb{P}$, neighs, *starter*) $\cup$ $\{p \in \mathbb{P} \mid \mathsf{idle.p}\}$ $\cup$ $\{p \in \mathbb{P} \mid \mathsf{father.p}\}$

assign

$[\!]_{p\in\mathbb{P}}$
$[\!]_{q\in\mathsf{neighs}.p}$ **if** idle.p $\wedge$ mit.$q.p$

then receive.$p.q.\langle$**process**$\rangle$ $\|$ father.$p := q$ $\|$ idle.$p :=$ false $\boxed{\| \text{IDLE\_a}_\Pi.p.q}$ (IDLE)

$[\!]$

$[\!]_{q\in\mathsf{neighs}.p}$ **if** $\neg$ idle.p $\wedge$ mit.$q.p$ $\wedge$ $collecting_\Pi.p.q$

then receive.$p.q.\langle$**process**$\rangle$ $\boxed{\| \text{COL\_a}_\Pi.p.q}$ (COL)

$[\!]$

$[\!]_{q\in\mathsf{neighs}.p}$ **if** $\neg$ idle.p $\wedge$ $can\_propagate.p.q$ $\wedge$ $propagating_\Pi.p.q$

then send.$p.q.\langle$**mes**$\rangle$ $\boxed{\| \text{PROP\_a}_\Pi.p.q}$ (PROP)

$[\!]$

if $finished\_col\_and\_prop.p \wedge \neg reported\_to\_father.p$

then send.$p.$(father.p).$\langle$**mes**$\rangle$ $\boxed{\| \text{DONE\_a}_\Pi.p}$ (DONE)

Figure 2: Augmented skeleton to deal with different communication strategies.

In order to adjust the skeleton from Figure 1 we synchronously superpose 4 functions upon it: IDLE_a$_\Pi$, COL_a$_\Pi$, PROP_a$_\Pi$ $\in$ $\mathbb{P} \to \mathbb{P} \to$ Action, and DONE_a$_\Pi$ $\in$ $\mathbb{P} \to$ Action, that specify additional functionality for each action in the skeleton. The result is in Figure 2. We can now complete the characterization of TARRY by defining for all $p, q \in \mathbb{P}$ that IDLE_a$_{\text{TARRY}}.p.q$ and COL_a$_{\text{TARRY}}.p.q$ equal le_rec.$p :=$ true, and that PROP_a$_{\text{TARRY}}.p.q$ and DONE_a$_{\text{TARRY}}.p$ equal le_rec.$p :=$ false. Note that IDLE_a.$p.q$, COL_a.$p.q$, PROP_a.$p.q$ and DONE_a.p for ECHO and DS are all skip.

In order to be able to formalize the DFS algorithm, we need to be able to remember the identity of the sender of the last incoming message. In order to make this possible, we introduce a new variable lp_rec.p (last process of which p has received a message) for every node p. Then we define for all $p \in \mathbb{P}$ that IDLE_a$_{\text{DFS}}.p.q$ and COL_a$_{\text{DFS}}.p.q$ equal le_rec.$p :=$ true $\|$ lp_rec.$p := q$, and that PROP_a$_{\text{DFS}}.p.q$ and DONE_a$_{\text{DFS}}.p$ are le_rec.p := false. Thus, we define:

$propagating_{\text{DFS}}.p.q = propagating_{\text{TARRY}}.p.q \wedge (q = \mathsf{lp\_rec}.p \ \vee \ \neg can\_propagate.p.(\mathsf{lp\_rec}.p))$

$collecting_{\text{DFS}}.p.q = collecting_{\text{TARRY}}.p.q$

$init_{\text{DFS}} = init_{\text{TARRY}}$

6 Applications of Diffusing Computations

Precise discussion of the applications of diffusing computations requires us to specify the contents of the messages that are being sent and the ways these are processed upon receipt. Consequently, we parameterize our skeleton such that specific applications can be defined as *instantiations* of the underlying algorithm, and abstraction from applications is done by *universal quantification* over its arguments. In order to make the algorithms suitable for the characterization of specific applications, they will have to be parameterized with additional parameters:

- a predicate $initA \in$ Expr, which can be used to specify an additional application specific initial condition.

$\Pi.initA.\text{IDLE\_r}.\text{COL\_r}.\text{PROP\_mes}.\text{DONE\_mes}.\text{IDLE\_a}.\text{COL\_a}.\text{PROP\_a}.\text{DONE\_a}.\mathcal{A}.\mathcal{V} =$

prog Π

init ASYNC_Init.($\mathbb{P}$, neighs, *starter*) $\wedge$ $init_\Pi$ $\wedge$ $\boxed{initA}$

var CV.($\mathbb{P}$, neighs, *starter*) $\cup$ $\{p \in \mathbb{P} \mid \mathsf{idle.p}\}$ $\cup$ $\{p \in \mathbb{P} \mid \mathsf{father.p}\}$ $\cup$ $\boxed{\mathcal{V}}$

assign

$[]_{p \in \mathbb{P}}$

$[]_{q \in \mathsf{neighs}.p}$ **if** $\mathsf{idle}.p \wedge \mathsf{mit}.q.p$ (IDLE)

then $\mathsf{receive}.p.q.\boxed{(\text{IDLE\_r}.p.q)} \parallel \mathsf{father}.p := q \parallel \mathsf{idle}.p := \mathsf{false} \parallel \text{IDLE\_a}_\Pi.p.q \parallel \boxed{\text{IDLE\_a}.p.q}$

$[]$

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge \mathsf{mit}.q.p \wedge collecting_\Pi.p$ (COL)

then $\mathsf{receive}.p.q.\boxed{(\text{COL\_r}.p.q)} \parallel \text{COL\_a}_\Pi.p.q \parallel \boxed{\text{COL\_a}.p.q}$

$[]$

$[]_{q \in \mathsf{neighs}.p}$ **if** $\neg\ \mathsf{idle}.p \wedge can\_propagate.p.q \wedge propagating_\Pi.p$ (PROP)

then $\mathsf{send}.p.q.\boxed{(\text{PROP\_mes}.p.q)} \parallel \text{PROP\_a}_\Pi.p.q \parallel \boxed{\text{PROP\_a}.p.q}$

$[]$

if $finished\_col\_and\_prop.p \wedge \neg reported\_to\_father.p$ (DONE)

then $\mathsf{send}.p.(\mathsf{father}.p).\boxed{(\text{DONE\_mes}.p)} \parallel \text{DONE\_a}_\Pi.p \parallel \boxed{\text{DONE\_a}.p}$

$[]$

$\boxed{\mathcal{A}}$

Figure 3: Parameterized skeleton to deal with various applications.

- a function IDLE_r $\in \mathbb{P} \to \mathbb{P} \to \mathsf{Expr} \to \mathsf{Action}$, that specifies how an idle node should process messages upon receipt.
- a function COL_r $\in \mathbb{P} \to \mathbb{P} \to \mathsf{Expr} \to \mathsf{Action}$, that specifies how a non-idle process should process messages upon receipt.
- a function PROP_mes$\in \mathbb{P} \to \mathbb{P} \to \mathsf{Expr}$, that, given a node p, specifies what message p should send to a neighbor in its propagating phase.
- a function DONE_mes$\in \mathbb{P} \to \mathsf{Expr}$, that, given a node p, specifies which message p finally has to send to its father.
- again 4 functions like: IDLE_a, COL_a, PROP_a$_\Pi$ $\in \mathbb{P} \to \mathbb{P} \to \mathsf{Action}$, and DONE_a $\in \mathbb{P} \to \mathsf{Action}$, that specify additional functionality by synchronous superposition for each action in the skeleton but are independent of the underlying algorithm.
- a set of actions $\mathcal{A}$ that, by ways of asynchronous superposition, can be used to add additional functionality or behavior.
- a set of new variables $\mathcal{V}$ that are assigned by the superposed (synchronous and asynchronous) actions.

Figure 3 shows the result. After discussing termination of diffusing computations in §7, we will show how easy it now becomes to create specifications and prove the correctness of specific applications of diffusing computations.

7 Termination of Diffusing Computations

Termination of diffusing computations means that when the algorithm is started in the initial state, eventually each node will reach the situation in which it neither sends nor receives any more messages, and all communication channels will be empty. Termination is independent of the contents of the messages that are being sent and how these are processed upon receipt. Suppose we have arbitrary *initA*, IDLE_r, COL_r, PROP_mes, DONE_mes, IDLE_a, COL_a, PROP_aand DONE_a. Let us abbreviate, for some $\Pi \in \{$DS, ECHO, PIF, TARRY, DFS$\}$:

$T_\Pi = \Pi.initA.\text{IDLE\_r}.\text{COL\_r}.\text{PROP\_mes}.\text{DONE\_mes}.\text{IDLE\_a}.\text{COL\_a}.\text{PROP\_a}.\text{DONE\_a}.\mathcal{A}$

Termination for this family T_Π of algorithms, for some invariant J_Π of Π, is specified as:

$$J_\Pi \ {}_{T_\Pi}\vdash \ \mathbf{ini}T_\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : done.p) \tag{1}$$

Evidently, when this specification has been proved for some invariant J_Π of Π, it can be inferred for all possible combinations of *initA*, IDLE_r, COL_r, PROP_mes, DONE_mes, IDLE_a, COL_a, PROP_a, DONE_a, and $\mathcal{A}$ that do not assign to the underlying variables of Π. As will be shown, this significantly reduces the proof effort for specific applications of our base algorithms. The proof of specification (1) can be found in [32].

8 Propagation of Information with Feedback

PIF (**P**ropagation of **I**nformation with **F**eedback), is the problem of broadcasting a piece of information I to all nodes in a connected network in such a way that all the nodes "know" when they have finished participating in the broadcast, one node in particular (the *starter*) "knows" that the broadcast is completed, and upon completion all nodes own the piece of information I [25]. A node p "knows" that it has finished participating in the broadcast when it has received and sent messages from and to all its neighbors, i.e. when *done.p*. Moreover, when the *starter* is *done* it "knows" that the broadcast is completed. To make Π suitable for the PIF application, we introduce new local variables $V.p$ for all processes $p \in \mathbb{P}$ and instantiate Π such that: (a) the starter initially has I stored in $V.starter$ **;** (b) upon receipt of a message m in the IDLE phase of node p, this value is copied directly to $V.p$ **;** (c) in the PROP phase of node p, the value stored in ($V.p$) is sent **;** (d) the contents of the messages received in the col phase are discarded **;** (e) the contents of the messages sent in the DONE phase can remain unspecified. Now PIF can be defined as:

$\text{PIF}_\Pi = \Pi.initA.\text{IDLE\_r}.\text{COL\_r}.\text{PROP\_mes}.\text{DONE\_mes}.\text{IDLE\_a}.\text{COL\_a}.\text{PROP\_a}.\text{DONE\_a}.\mathcal{A}.\mathcal{V}$

where DONE_mes is arbitrary, IDLE_a = COL_a = PROP_a = DONE_a = skip, $\mathcal{A} = \emptyset$, $\mathcal{V} = \{p \in \mathbb{P} \mid V.p\}$, and:

$initA \ = \ (\lambda s \cdot ((s \circ V).starter) = I)$
IDLE_r$= (\lambda p, q, m \cdot V.p := m)$
COL_r$= (\lambda p, q, m \cdot \mathsf{skip})$
PROP_mes$= (\lambda p, q \cdot V.p)$

The formal specification of PIF applications reads:

$$(J_\Pi \wedge J_{\text{PIF}}) \ {}_{\text{PIF}\_\Pi}\vdash \mathbf{ini}\text{PIF}_\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : done.p) \ \wedge \ (\forall p : p \in \mathbb{P} : (V.p) = I) \tag{2}$$

where J_{PIF} is an invariant of PIF_Π stating additional safety behavior. Since PIF_Π is an instantiation of Π, the correctness criterion above can be reduced:
(2)
$\Leftarrow$ (Thm.3)
$(J_\Pi \wedge J_{\text{PIF}}) \ {}_{\text{PIF}_\Pi}\vdash \mathbf{ini}\text{PIF}_\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : done.p)$
$(J_\Pi \wedge J_{\text{PIF}}) \ {}_{\text{PIF}_\Pi}\vdash \forall p : p \in \mathbb{P} : done.p \rightsquigarrow \forall p : p \in \mathbb{P} : (V.p) = I$

The first conjunct is implied by specification (1) (use Thm.1, Thm.2, and the fact that $\mathbf{ini}\text{PIF}_\Pi$ includes $\mathbf{ini}\Pi$). The second conjunct is application specific and, using Thm.4 and Thm.6, can be reduced to:
${}_{\text{PIF}_\Pi}\vdash \circlearrowleft (J_\Pi \wedge J_{\text{PIF}})$
$\forall p : p \in \mathbb{P} : (J_\Pi \wedge J_{\text{PIF}} \wedge done.p \Rightarrow (V.p) = I)$
The only thing left to do is find the characterization of the invariant J_{PIF} that establishes the last conjunct. The most straightforward and evident choice is: $(\forall p : p \in \mathbb{P} : done.p \Rightarrow (V.p) = I)$. This is obviously a stable property, and sufficient to prove the conditions stated above.

9 Computation of Summation

Suppose that every node $p \in \mathbb{P}$ in the network has a unique local variable that stores some data value, and that the distribution of these local variables is given by a function $V : \mathbb{P} \rightarrow \texttt{Var}$. Imagine $D : \mathbb{P} \rightarrow \texttt{Val}$ being a snapshot of the values that reside at the local variables in some state s, i.e. $D = s \circ V$.

It is straightforward [8, 17, 18, 29, 30] to instantiate our algorithm such that the sum of all data values is computed and eventually resides at the *starter* (i.e. is stored in the variable $V.starter$). Instantiate Π such that: D_i is defined to be the initial distribution of the data values**;** in the (PROP) phase of node p, the identity element of $+$ (say 0) is sent**;** in the (DONE) phase of node p, $(V.p)$ is sent**;** upon receiving a message m in the (IDLE) and (COL) phase, this value m is added to the data value that resides in $(V.p)$, and the result is stored in $(V.p)$. We define the specific SUM-application of Π by

$$\mathrm{S}_\Pi = \Pi.initA.\text{IDLE\_r}.\text{COL\_r}.\text{PROP\_mes}.\text{DONE\_mes}.\text{IDLE\_a}.\text{COL\_a}.\text{PROP\_a}.\text{DONE\_a}.\mathcal{A}.\mathcal{V}$$

where IDLE_a=COL_a=PROP_a=DONE_a=skip, $\mathcal{A} = \emptyset$, $\mathcal{V} = \{p \in \mathbb{P} \mid V.p\}$ and, for a commutative monoid $(+, 0)$:

$initA = (\lambda s \cdot D_i = (s \circ V))$
IDLE_r = COL_r $= (\lambda p, q, m \cdot V.p := V.p \ + \ m)$
PROP_mes $= (\lambda p, q \cdot 0)$
DONE_mes $= (\lambda p \cdot V.p)$

The formal specification of these summation algorithms reads:

$$J_\Pi \wedge J_S \ {}_{\mathrm{S}_\Pi}\vdash \ \mathbf{ini}\mathrm{S}_\Pi \rightsquigarrow (V.starter) = \sum_{x \in \mathbb{P}} D_i.x \tag{3}$$

where J_S is an invariant stating additional safety behavior of S_Π. We now reduce the correctness criterion for S_Π in a way that progress properties already proved for Π are inherited and we only need to prove application specific properties.

(3)
$\Leftarrow$ (Thm.1, Thm.2, **ini**S_Π includes **ini**Π)
$J_\Pi \ {}_{\mathrm{S}_\Pi}\vdash \ \mathbf{ini}\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : done.p)$
${}_{\mathrm{S}_\Pi}\vdash \circlearrowleft (J_\Pi \wedge J_S)$
$J_\Pi \wedge J_S \wedge (\forall p : p \in \mathbb{P} : done.p) \Rightarrow \ V.starter = \sum_{x \in \mathbb{P}} D_i.x$

The first conjunct equals (1). Consequently, we are left with finding the characterization of invariant J_S that establishes the last conjunct. Although some ingenuity is required in order to come up with this invariant, its construction is guided by the availability of information about its use within the process of verifying the above correctness criterion. Any message sent during the execution of the algorithm is: either 0 (in the PROP phases), or the data value $(V.p)$ residing at some node p (in the DONE phases). However, after sending $(V.p)$, node p will be *done*. Hence, since 0 is the identity element of $+$, the desired sum SUM will, in any state, be: the sum of values that reside at the nodes that are not *done*, added to the sum of values that are in transit in s. The safety property J_S in state s:

$$\begin{aligned}\textstyle\sum_{p \in \mathbb{P}} D_i.p \ = & (s \circ V).starter \\ & + \textstyle\sum_{p \in \mathbb{P} \wedge (p \neq starter) \wedge \neg done.p} (s \circ V).p \\ & + (\text{the values in the communication channels in } s)\end{aligned}$$

With this definition we can prove the required application specific requirements from above. Note that invariant J_Π is still unspecified at this point, and its precise characterization is not needed to be able to derive a proof strategy for the summation application. Consequently, any instantiation of the DS, ECHO, PIF, TARRY and DFS algorithms that maintains safety property J_S can be used to compute the sum. This clear separation of progress and safety properties enables us to prove the correctness of any application by verifying the safety property of the application and inheriting the progress proof (including invariant J_Π that was needed to prove this progress) of the underlying algorithm.

10 Mattern's Four Counter Solution for Termination Detection

The problem of termination detection is to superimpose a *control algorithm* on a given so-called *basic computation* such that the first can detect when the termination condition holds for the latter.

In [20] Mattern proposes a termination detection solution called the Four Counter Method (FCS). The solution departs from a basic computation with atomic actions, arbitrary message delays and processes (taken from a set $\mathbb{P}$) that are always "passive"", meaning that a process may at any time take any message from one of its incoming communication channels, change it its local state and send out any number of messages. Such a distributed computation is considered to be *terminated* if there are no messages in transit. In order to be able to test this condition, every process $p \in \mathbb{P}$ of the basic computation keeps a counter $s.p$ for the number of sent basic messages, and a counter $r.p$ for the number of received basic messages. The control algorithm visits all the processes during a first (red) round to accumulate the counters by $\mathcal{S} = \sum_{p\in\mathbb{P}}\ s.p$ and $\mathcal{R} = \sum_{p\in\mathbb{P}}\ r.p$. Then a second (green) round is started after completion of the first, yielding the accumulated counter values $\mathcal{S}'$ and $\mathcal{R}'$. In [20] it is proved that the termination condition (i.e. there are no messages in transit) holds if $\mathcal{S} = \mathcal{R} = \mathcal{S}' = \mathcal{R}'$.

The two rounds of this control algorithm can easily be implemented by any diffusing computation. The red round is when all processes get non-idle, the green round is when all processes become done. In order to capture the current values of $s.p$ and $r.p$ at both rounds, we introduce four variables for each process $p \in \mathbb{P}$ in the control algorithm: $s_{red}.p$, $s_{green}.p$, $r_{red}.p$, $r_{green}.p$. Moreover, we assign $s_{red}.p$, $r_{red}.p := s.p$, $r.p$ when p becomes non-idle, and $s_{green}.p$, $r_{green}.p := s.p$, $r.p$ when p becomes done. To accumulate the counter values, we introduce, for each process $p \in \mathbb{P}$, four variables $S.p$, $S'.p$, $R.p$, and $R'.p$ that respectively store the intermediate sum of the $s_{red}.q$, $s_{green}.q$, $r_{red}.q$, and $r_{green}.q$ values of those processes q that reside in the subtree that is rooted at p. Consequently, the rooted spanning tree that is created by the diffusing computation guarantees that, at the end of the control algorithm when the *starter* is done, it holds that:

$\mathcal{S} = \sum_{p\in\mathbb{P}}\ s_{red}.p = S.starter\ + s_{red}.starter$,
$\mathcal{R} = \sum_{p\in\mathbb{P}}\ r_{red}.p = R.starter\ + r_{red}.starter$,
$\mathcal{S}' = \sum_{p\in\mathbb{P}}\ s_{green}.p = S'.starter\ + s_{green}.starter$,
$\mathcal{R}' = \sum_{p\in\mathbb{P}}\ r_{green}.p = R'.starter\ + r_{green}.starter$,

and hence the *starter*, when it is done, can determine whether the termination condition holds. For this we will introduce a *new* variable term, that will be assigned by the *starter* when its done.

In the previous algorithms and their applications, the *starter* spontaneously started its execution by becoming non-idle. In the case of the FCS the starter should not only become non-idle, but also initialize $s_{red}.starter$ and $r_{red}.starter$ to $s.starter$ and $r.starter$ respectively. As before, we reflect this within the initial condition (**init**) of the algorithm. Consequently, we specify de algorithms as follows:

$\mathrm{FCS}_\Pi = \Pi.initA.\textsc{idle\_r}.\textsc{col\_r}.\textsc{prop\_mes}.\textsc{done\_mes}.\textsc{idle\_a}.\textsc{col\_a}.\textsc{prop\_a}.\textsc{done\_a}.\mathcal{A}.\mathcal{V}$

where $\textsc{idle\_r} = \textsc{col\_a} = \textsc{prop\_a} = \mathsf{skip}$, and:

$initA = \forall p \in \mathbb{P} : S.p = R.p = S'.p = R'.p = 0 \wedge s_{red}.starter = s.starter\ \wedge\ r_{red}.starter = r.starter$
$\textsc{idle\_a} = s_{red}.p, r_{red}.p := s.p, r.p$
$\textsc{col\_r} = \lambda p, q, m \cdot$ if $m =$<green $s\ s'\ r\ r'$>
then $S.p, R.p := S.p + s, R.p + r \parallel S'.p, R'.p := S'.p + s', R'.p + r'$
$\textsc{prop\_mes} = \lambda p, q \cdot$ <red>
$\textsc{done\_mes} = \lambda p, q \cdot$ <green $(S.p + s_{red}.p)\ (S'.p + s.p)\ (R.p + r_{red}.p)\ (R'.p + r.p)$>
$\textsc{done\_a} = s_{green}.p, r_{green}.p := s.p, r.p$
$\mathcal{A} = \{$ if $done.starter$
then term $:= (S.starter\ + s_{red}.starter) = (S'.starter\ + s.starter)$
$= (R.starter\ + r_{red}.starter) = (R'.starter\ + r.starter)$
$\parallel s_{green}.starter := s.starter \parallel r_{green}.starter := r.starter\ \}$
$\mathcal{V} = \{\{s_{red}.p, s_{green}.p, r_{red}.p, r_{green}.p\} \mid p \in \mathbb{P}\} \cup \{\{S.p, S'.p, R.p, R'.p\} \mid p \in \mathbb{P}\} \cup \{\mathsf{term}\}$

Note that in DONE_mes and $\mathcal{A}$ we use the s and r values instead of s_{green} and r_{green}. We have to do this because UNITY does not have sequential composition of assignment statements. However, since the assignment statements are executed in parallel, the result is the same.

The formal specification of these algorithms reads:

$$J_\Pi \wedge J_{\textsc{fcs}\ \mathrm{FCS}_\Pi} \vdash \mathbf{ini}\mathrm{FCS}_\Pi \rightsquigarrow \mathsf{term} \equiv \sum_{p\in\mathbb{P}} s_{red}.p = \sum_{p\in\mathbb{P}} s_{green}.p = \sum_{p\in\mathbb{P}} r_{red}.p = \sum_{p\in\mathbb{P}} r_{green}.p \qquad (4)$$

where $J_{\textsc{fcs}}$ is an invariant stating additional safety behavior of FCS_Π. Again, we can reduce the correctness criterion for in a way that progress properties already proved for Π are inherited and we only need to prove application specific properties.

(4)
$\Leftarrow$ (Thm.5, Thm.1, Thm.2, $\mathbf{ini}\mathrm{FCS}_\Pi$ includes $\mathbf{ini}\Pi$)

$J_\Pi \ {}_{\text{FCS}\Pi}\vdash \ \mathbf{ini}\Pi \rightsquigarrow (\forall p : p \in \mathbb{P} : \mathit{done}.p)$
${}_{\text{FCS}\Pi}\vdash \circlearrowleft (J_\Pi \wedge J_{\text{FCS}})$
$J_\Pi \wedge J_{\text{FCS}} \ {}_{\text{FCS}\Pi}\vdash (\forall p : p \in \mathbb{P} : \mathit{done}.p) \rightsquigarrow \mathsf{term} \equiv \sum_{p\in\mathbb{P}} s_{red}.p = \sum_{p\in\mathbb{P}} s_{green}.p = \sum_{p\in\mathbb{P}} r_{red}.p = \sum_{p\in\mathbb{P}} r_{green}.p$

Again, the first conjunct can be proved by (1), since the new assignments superimposed on Π do not write to Π's underlying variables. The last conjunct, using Thm.4, can be refined to

$$J_\Pi \wedge J_{\text{FCS}} \wedge (\forall p : p \in \mathbb{P} : \mathit{done}.p) \ \mathsf{ensures} \ \mathsf{term} \equiv \sum_{p\in\mathbb{P}} s_{red}.p = \sum_{p\in\mathbb{P}} s_{green}.p = \sum_{p\in\mathbb{P}} r_{red}.p = \sum_{p\in\mathbb{P}} r_{green}.p$$

This one-step progress properties is satisfied by the action in $\mathcal{A}$, and so we are left with finding the characterization of invariant J_{FCS} that enables us to prove this. Since four summations are being calculated simultaneously, we will have four invariants that are similar to the invariant of the summation algorithm from the previous section. More specifically, J_{FCS} will be defined for all quadruples $(X, c, x, n) \in \{(S, red, s, 1), (S', green, s, 2), (R, red, r, 3), (R', green, r, 4)\}$ as:

$$\begin{aligned}\textstyle\sum_{p\in\mathbb{P}} x_c.p = {} & X.starter + x_c.starter \\ & + \textstyle\sum_{p\in\mathbb{P}\wedge(p\neq starter)\wedge\neg done.p} X.p + x_c.p \\ & + (\text{the } n\text{th-value of the green messages in transit})\end{aligned}$$

With this definition of J_{FCS}, we can prove the required application specific requirements from above.

11 Conclusion

We have given a highly parameterized abstract algorithm for diffusing computations, and we have shown that many existing algorithms and their applications can be obtained from this abstract algorithm by instantiating the parameters appropriately. By clearly separating the progress and safety properties of the different applications, we can proof the correctness of any application by verifying an application-specific safety property and inheriting the already proved termination and safety of the underlying algorithm.

Studies with postgraduate students have given empirical evidence that our re-usable specifications and proofs improve their understanding of the algorithms, increase the slope of the learning-curve, and enable students to prove the correctness of some specific applications within a remarkably short amount of time.

Theorems About $\circlearrowleft$ and $\rightsquigarrow$

Thm 1 $\rightsquigarrow$ *Substitution:* $\dfrac{(J \wedge p \Rightarrow q) \wedge \ J \ {}_P\vdash (q \rightsquigarrow r) \wedge (J \wedge r \Rightarrow s)}{J \ {}_P\vdash p \rightsquigarrow s}$

Thm 2 $\rightsquigarrow$ *Stable Strengthening:* $\dfrac{({}_P\vdash\circlearrowleft (J_1 \wedge J_2)) \ \wedge \ J_1 \ {}_P\vdash p \rightsquigarrow q}{(J_1 \wedge J_2) \ {}_P\vdash p \rightsquigarrow q}$

Thm 3 $\rightsquigarrow$ *Accumulation:* $\dfrac{(J \ {}_P\vdash p \rightsquigarrow q) \ \wedge \ (J \ {}_P\vdash q \rightsquigarrow r)}{J \ {}_P\vdash p \rightsquigarrow q \wedge r}$

Thm 4 $\rightsquigarrow$ *Introduction:* $\dfrac{\circlearrowleft (J \wedge q) \wedge ([J \wedge p \Rightarrow q] \ \vee \ (p \wedge J \ \mathsf{ensures} \ q))}{J \ {}_P\vdash p \rightsquigarrow q}$

Thm 5 $\rightsquigarrow$ *Trans:* $\dfrac{J \ {}_P\vdash (p \rightsquigarrow q) \ \wedge \ J \ {}_P\vdash (q \rightsquigarrow r)}{J \ {}_P\vdash p \rightsquigarrow r}$

Thm 6 $\rightsquigarrow$ *Conjunction:* $(W \neq \emptyset)$ $\dfrac{(\forall i : i \in W : J \ {}_P\vdash p.i \rightsquigarrow q.i)}{J \ {}_P\vdash (\forall i : i \in W : p.i) \rightsquigarrow (\forall i : i \in W : q.i)}$

References

[1] A. Arora and M.G. Gouda. "Distributed Reset" *IEEE Trans. on Comp.*, 43(9):1026–1038, 1994.

[2] K.M. Chandy and J. Misra. "Distributed computations on graphs" *Com. ACM*, 25(11):833–838, 1982.

[3] K.M. Chandy and J. Misra, "Termination detection of diffusing computations in communicating sequential processes" *ACM Tr. Prog. Lang. and Sys.* 4(1):37–43, 1982.

[4] K.M. Chandy and J. Misra. "Distributed deadlock detection." *ACM Tr. Comp. Sys.* 1(2):144–156, 1983.

[5] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation*, AW, 1988.

[6] E.J.H. Chang. "Echo algorithms" *IEEE Trans. Softw. Eng.*, (SE-8):391–401, 1982.

[7] T.-Y. Cheung. "Graph traversal techniques and the maximum flow problem in distributed computation" *IEEE Tr. on Softw. Eng.*, SE-9(4):504–512, 1983.

[8] C.-T. Chou. "Mechanical verification of distributed algorithms in higher order logic" *Proc. 7th Workshop on HOLTP*, T. Melham and J. Camilleri (eds) *LNCS* 859, 1994.

[9] C.-T. Chou. "Using operational intuition about events and causality in assertional proofs." TR 950013, UCLA, Feb. 1995.

[10] A. Cournier, A. Datta, F. Petit, and V. Villain. "Self-Stabilizing PIF algorithm in arbitrary rooted networks." *21th ICDCS*, pages 91–98, 2001.

[11] E.W. Dijkstra and C.S. Scholten. "Termination detection for diffusing computations." *Information Processing Letters*, 11(1):1–4, 1980.

[12] S.G. Finn. "Resynch procedures and a fail-safe network protocol." *IEEE Trans.*, COM-27:840–845, 1979.

[13] M. Filali, P. Mauran, G. Padiou, P. Quinnec, X. Thirioux. "Refinement based validation of an algorithm for detecting distributed termination." *FMPPTA2000*, LNCS 1800, 2000.

[14] N. Francez. "Distributed termination" *ACM Trans. Prog. Lang.*, 2(1):42–55, 1980.

[15] J.J. Garcia-Lunes-Aceves. "Loop-free routing using diffusing computations." *IEEE/ACM Trans. on Netw.*, 1(1):130–141, 1993.

[16] M.J.C. Gordon and T.F. Melham. *Introduction to HOL.* CUP, 1993.

[17] J. Groote, F. Monin, and J. Springintveld. "A computer checked algebraic verification of a distributed summation algorithm." TR-CSR-97-14, TUE, 1997.

[18] W.H. Hesselink. "A mechanical proof of Segall's PIF algorithm." TR-CS-R9604, UG, 1996.

[19] S. Huang. "A Distributed Deadlock Detection Algorithm for CSP-Like Communication," *ACM Tr. on Prog. Lang. and Sys*, 12(1):102–122, 1990.

[20] F. Mattern. "Algorithms for Distributed Termination Detection." *Distributed Computing*, vol 2, pp 161–175, 1987.

[21] F. Mattern. "Distributed Control Algorithms (Selected Topics)." *Parallel Computing on Distributed Memory Multiprocessors*, F. Özguner, F. Ercal (Eds.), pp. 167–185, 1993.

[22] L. Onana Alima. *Self.stabilization by self-stabilizing waves.* UCL 2000.

[23] W. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms.* UU 1995.

[24] M. Raynal and J-M. Helary. *Synchronisation and control of distributed systems.* 1990.

[25] A. Segall. "Distributed network protocols." *IEEE Trans. IT*, (IT-29):23–35, 1983.

[26] F.A. Stomp. *Design and verification of Distributed Network Algorithms: Foundations and Applications.* PhD thesis, TUE, 1989.

[27] M. G. Tarry. "Le problème des labyrinthes." *Nouvelles Annales de Mathématique*, (149):187–190, 1895.

[28] G. Tel. *The Structure of Distributed Algorithms.* PhD thesis, UU, 1989.

[29] G. Tel. *Introduction to Distributed Algorithms.* CUP, 1994.

[30] F.W. Vaandrager. Verification of a distributed summation algorithm. CS-R9505, CWI 1995

[31] T.E.J. Vos and S.D. Swierstra. "Program refinement in UNITY" UU-CS-2001-41, 2001.

[32] T.E.J. Vos and S.D. Swierstra. "Proving distributed hylomorphisms" UU-CS-2001-40, 2001.