Formal Design of Self-stabilizing Programs: Theory and Examples

I.S.W.B. Prasetya, S.D. Swierstra*

Abstract

It is commonly realized that informal reasoning about distributed algorithms in general and self-stabilizing systems in particular is very error-prone. Formal method is considered as a promising solution, but is still in an immature state for the fact that formal proofs of even simple algorithms are tedious and difficult to follow. We believe that to make the method more appealing one should not only pay attention to 'theoretical' issues such as consistency and completeness, but also to the 'ergonomics' of the method. In this spirit, this paper proposes a number of new operators to model self-stabilization and a formalization of a number of useful design strategies. It is hoped that their use can improve the ease with which algorithmic reasoning is formally applied. Some examples showing how various laws are used will given in Part II.

*Universiteit Utrecht, Instituut Informatica, Postbus 80.089, 3508 TB Utrecht, Nederland. Email: wishnu@cs.uu.nl, doaitse@cs.uu.nl

Part I
Theory

1 Introduction

The concept of *self-stabilization* was first conceived by E.W. Dijkstra [9]. A self-stabilizing program is a program that will reach and remain in a set of predefined states —the so-called *legal states*— regardless of its initial state. For a distributed system such a property is very desirable since it has the ability to, given enough time, recover from any perturbation (be that a failure or an update sent by the system's environment) without any outside intervention. Since the work of Dijkstra there have been many papers addressing the topic, for example [13, 6, 5], and many self-stabilizing algorithms have been invented, for example [4, 8, 14].

Reasoning about self-stabilization is often complicated. Most people are aware of various design strategies, yet applying them formally can suddenly be an entirely different experience. It was not until recently that people attempt to deal with self-stabilization more formally. The first to formalize the idea are Arora and Gouda [4]. However, reasoning is still carried out informally. A step forward is made by Herman [12] by proposing a number composition laws of stabilization. A truly formal treatment is later given by Lenfert and Swierstra [15] using a programming logic called UNITY. They prove various calculational properties of stabilization. This paper presents a further development of the latter work by adding new more versatile operators to express program behavior and and by formalizing a number of useful design strategies. For example, we add laws concerning a decomposition technique called *layering* –a useful technique in self-stabilization [12, 3]– and laws capturing round-wise stabilization

(which is comparable to loops in sequential programming). Several examples demonstrating how the laws are exercised is given in Part II.

Our approach is based on UNITY, a programming language and logic by Chandy and Misra [7]. UNITY is simple, but powerful enough to model distributed systems. In addition, all theorems mentioned in this paper have been mechanically verified using a proof assistant called HOL[1]. The theorems are available in the form of a library and can be re-used. The complete package containing the theorems and their proof scripts is available at request. As for UNITY itself, it has also been mechanically verified in HOL by Anderssen [2] and Prasetya [19].

Sections overview: Section 2 provides some informal motivation. Section 3 explains the notation. Section 4 gives a brief review on UNITY and introduces an extension used in this work. Section 5 discusses how the notion self-stabilization is formalized and provides a set of laws to deal with it. Section 6 discusses how progress and stabilization behave with respect to parallel composition. Section 7 discusses inductive decomposition. Finally, Some conclusions are provided in Section 8.

2 Motivation

This paper presents a formalization of two important techniques in formal design of self-stabilizing systems, namely *parallel composition* and *inductive stabilization.*

2.1 Parallel Composition

Parallel composition has always been a though issue in distributed computing. Especially progress properties are known to be destroyed by parallel composition. Given that, a practically interesting issue is to know under what condition we can be sure that progress is preserved by the composition; or, even if it is destroyed, in what way exactly a progress property is destroyed. These are the issues that will be highlighted in this paper.

In general, the ability to decompose a global specification into specifications of component programs is often referred to as *compositionality.* This would enable us to design each component in isolation (thus supporting the so-called *modular design approach*) and may reduce the amount of proof obligations. To be able to do this kind of decomposition we need laws of the form:

$$\frac{(P \text{ sat spec1}) \ \wedge \ (Q \text{ sat spec2})}{P \otimes Q \text{ sat } (\text{spec1} \oplus \text{spec2})} \tag{2.1}$$

[1] HOL is a software system, developed by M. Gordon, to interactively write (and check) a proof. The system is based on a higher order logic. The soundness of the system is guaranteed in the sense that no false theorem can be generated. The system is extensible and provides a whole range of highly programmable proof-tools. For a description of HOL, see [10].

where P and Q are programs, $\otimes$ is some kind of program composition, and spec1 and spec2 are specifications. The theorem describes how a property of $P \otimes Q$ can be split to specifications of P and Q. In particular, we are interested in the case where $\otimes$ is the parallel composition ($[\!]$).

To illustrate some of the problems encountered consider the following example. Let ${}_P\vdash p \rightarrowtail q$ mean that if p holds during an execution of P then eventually q will hold. So, $\rightarrowtail$ describes progress.

Let a, b, and c be boolean variables. Suppose now ${}_P\vdash a \rightarrowtail c$ holds. The property does not refer to b, so we may expect that if we put P in parallel with Q defined below then the progress will be preserved.

Q: **do forever** $b := \neg b$

However, even though the expression ${}_P\vdash a \rightarrowtail c$ does not refer to b, it may happen that the progress actually depends on b, for example if P is the following program:

P: **do forever** { **if** a **then** $b :=$ true ; **if** b **then** $c :=$ true }

In this case, Q will destroy the progress $a \rightarrowtail c$. Still, if we put P in parallel with Q' which only writes to b under the condition, say, C (and does nothing to a and c) we can still conclude that the composite program will have the property $a \wedge C \rightarrowtail c \vee \neg C$. In fact, this is an instance of a very generic composition property of progress observed by Singh [22].

The example suggests that recording the set of variables upon which a progress property depends may enable us to draw useful compositionality results. The work of Udink, Herman, and Kok [23] is a full justification of this. In this paper we will take a simpler approach. We observe that the only part of a program that is ever influenced by its own actions is its write variables. Compositionality is achieved by: (1) restricting progress specification ${}_P\vdash p \rightarrowtail q$ to only describe progress made on the write variables, and (2) adding a forth parameter J which is a stable predicate in P and may describe assumed values of read-only variables. So, a progress specification now looks like $J\ {}_P\vdash p \rightarrowtail q$. The progress described is not from p to q, but from $J \wedge p$ to q. This way of modelling progress turns out to be sufficient to get the general progress composition property predicted by Singh, and yet results in a formalism which is more workable than the one proposed by Udink, Herman, and Kok.

Especially interesting results can be obtained for programs that are *write-disjoint* —that is, programs that do not share any write variable. Let P and Q be two write-disjoint programs. Suppose $J\ {}_Q\vdash p \rightarrowtail q$ holds. Since P and Q are write-disjoint the only way P can influence Q is by writing to Q's read-only variables. J already assumed under what condition of read-only variables the progress from $p \wedge J$ to q can be made by Q. Consequently, if P also respects the assumption made on the read-only variables of Q, that is, P respects the stability of J, then the progress can be maintained in $P[\!]Q$ (parallel composition of P and Q). This principle is called *transparency* principle. It can be expressed

as follows:

$$\frac{J \text{ is stable in } P \ \wedge \ (J \ {}_Q\!\vdash p \rightarrowtail q)}{J \ {}_{P\|Q}\!\vdash p \mapsto q} \quad \text{if } P \text{ and } Q \text{ are write-disjoint} \qquad (2.2)$$

Note that the law has the form of (2.1). A formal treatment of this kind of laws will be given in Section 6.

Composition of write-disjoint programs occurs frequently in practice. Trivial examples are program that share no variable or programs that only share read-only variables. If we have a program P which writes to the read-only variables of Q whereas Q does not write back to P then we have a construction that in [3] is called *layering*. Layering is a useful technique. For example, to detect termination we can built the detection program as a layer 'on top' the actual program.

2.2 Inductive Stabilization

A self-stabilizing system typically relies on some inductive stabilization strategy to reach its goal. The strategy can be simple, or, as in quite many cases, it can be complicated [1, 8, 14]. This strategy is important because it is the heart of the system, and consequently also the center around which the correctness proof of the system is built. Typically, well-founded induction on *progress* is used to formally capture the inductiveness. But stabilization is a stronger property than progress, so one may conjecture that a stronger, more specialized, version of induction exists for stabilization. Such an induction will be more effective to handle inductive stabilization. Lentfert and Swierstra propose such an '*inductive stabilization theorem*' which they call *round decomposition* [15], but the formulation is rather complicated, which is because they make their theorem too special purpose. We propose a weaker, more general, formulation of round decomposition. Because it is more general, it is also simpler and nicer. The round decomposition theorem of Lentfert and Swierstra can still be derived from our theorem [20].

3 Notation

Functions

The *application* of a function f to x is written as $f.x$. To improve readability, the 'dot' denoting the application of some functions is made 'disappear' like in the application of $\mathbf{w}$ in $\mathbf{w}P$.

Sets

The set notation used is standard except perhaps the following. The *restriction* of $f \in A \rightarrow B$ with respect to a set $S \subseteq A$ is written $f \restriction S$ and is a function of type $S \rightarrow B$ with the following property:

$$(\forall x : x \in S : (f \restriction S).x = f.x) \qquad (3.1)$$

Set complement is denoted by a superscript c like in S^c. The *power set* of a set S is denoted by $\mathcal{P}(S)$. *Set abstraction* is written as $\{x : P.x : f.x\}$ instead of the usual $\{f.x|P.x\}$.

Predicates

A predicate over a set A is a function of type $A \rightarrow \mathsf{bool}$. For a predicate p over A, p is said to *hold everywhere*, denoted by $[p]$, iff $(\forall s : s \in A : p.s)$ holds.

Predicates which are used to describe the states of a program are called *state-predicates. Throughout this paper we will assume a universe of all available program variables, denoted with* Var*, and a universe of values, denoted by* Val. A *program-state* is a function of type $\mathsf{Var} \rightarrow \mathsf{Val}$. In a state s, the value of a variable x is given by $s.x$. A state-predicate is a predicate over program-states, so it has the type $(\mathsf{Var} \rightarrow \mathsf{Val}) \rightarrow \mathsf{bool}$.

For example $(\lambda s.\ 0 < s.x)$ is a state-predicate describing those states in which the value of x is greater than 0. It is a common practice that people write expressions like:

"$0 < x$", "$p \wedge q$", or "$(\exists i : P.i : x.i = 0)$"

in program specifications —for example as in "$\{0 < x\}\ x := x+1\ \{1 < x\}$"— to actually mean the corresponding state-predicates, which are, in the same order:

"$(\lambda s.\ 0 < s.x)$", "$(\lambda s.\ p.s \wedge q.s)$", and "$(\lambda s.\ (\exists i : P.i : s.(x.i) = 0))$"

Such overloading of symbols usually causes no confusion and since people are already used to it we will also do it. However, sometimes overloading does create confusion, so it helps if the reader is continuously aware of the fact that overloading is being used. The table below shows the 'lifted' meaning of the boolean operators true, $\neg$, $\wedge$, and $\forall$. Note that the dummy s ranges over program states. Other operators such as false, $\Rightarrow$, $\vee$, and $\exists$ can be derived from them.

Notation	Meaning
true	$(\lambda s.\ \mathsf{true})$
$\neg p$	$(\lambda s.\ \neg p.s)$
$p \wedge q$	$(\lambda s.\ p.s \wedge q.s)$
$(\forall i : P.i : p.i)$	$(\lambda s.\ (\forall i : P.i : p.i.s))$

A state-predicate p is said to be *confined* by a set of variables V if p is everywhere false or everywhere true or it does not restrict the value of any variable outside V. The reason for introducing the notion of confinement is that because we do not use partial functions in our logic. 'p is *confined* by V' is the same as saying p is a partial state-predicate over V. We will denote it with $p \in \mathsf{Pred}.V$.

Definition 3.1 : CONFINEMENT

$$p \in \mathsf{Pred}.V \ = \ (\forall s, t :: (s \restriction V = t \restriction V) \Rightarrow (p.s = p.t))$$

For example, $x+1 < y$ is confined by $\{x, y\}$ but not by $\{x\}$. true and false are confined by any set. Confinement is preserved by (standard) predicate operators mentioned above. So, for example, if $p, q \in \mathsf{Pred}.V$ then $p \wedge q \in \mathsf{Pred}.V$. Roughly one can say that if p is confined by V then p does not care about variables outside V. The reader may rightly observe that a predicate p is always confined by the set of all its free variables, but note that this set is not necessarily the smallest one confining p. For example, $\emptyset$ confines "$0 = x \vee 0 \neq x$". Another useful property is monotonicity:

Theorem 3.2 : CONFINEMENT MONOTONICITY

$$V \subseteq W \Rightarrow \mathsf{Pred}.V \subseteq \mathsf{Pred}.W$$

3.1 Binding Power

Figure 1 shows the relative binding power of the operators used in this paper. They are listed top to bottom in decreasing order of binding power with those listed in the same line bind equally strong.

"."
"**w**", "**r**", "**ini**", "**a**", "**ro**"
"c", "$\upharpoonright$"
"$\cap$", "$\cup$"
"$\in$", "$\subseteq$"
"$\neg$"
"$\wedge$", "$\vee$"
"$\Rightarrow$"
other operators
"$=$", "$\neq$"

Figure 1: Binding power of the operators.

---

4 A Brief Review on UNITY

The programming logic that we are going to use in reasoning about self-stabilizing programs is based on UNITY. UNITY is simple, yet powerful enough to allow reasoning about safety and progress properties of distributed programs. It views a program as a collection of *atomic* actions running in parallel. Parallelism is modelled by interleaving, and execution is infinite. So, an execution of a UNITY program is modelled by an infinite sequence of states in which at each step an action is selected and executed. There is no rule in the choice of action except

for this *fairness* rule: *every action in a UNITY program must be executed infinitely often.* For the fairness condition to make sense, it must be *assumed* that actions *always terminate.* For applications requiring things like action sequencing and termination, they can be modelled in UNITY by respectively location variables ala [16] and a predicate that continues to hold after sometime. Below is the variant of UNITY syntax used in this paper.

$$\begin{array}{lll} \langle \textit{Unity Program} \rangle & ::= & \textbf{prog}\ \langle \textit{name of program} \rangle \\ & & \textbf{read}\ \langle \textit{set of variables} \rangle \\ & & \textbf{write}\ \langle \textit{set of variables} \rangle \\ & & \textbf{init}\ \langle \textit{predicate} \rangle \\ & & \textbf{assign}\ \langle \textit{actions} \rangle \end{array}$$

actions is a list of *action* separated by $[]$. An *action* is either a single action or a set of indexed actions.

$$\begin{array}{lll} \langle \textit{actions} \rangle & ::= & \langle \textit{action} \rangle \mid \langle \textit{action} \rangle [] \langle \textit{actions} \rangle \\ \langle \textit{action} \rangle & ::= & \langle \textit{single action} \rangle \mid ([] i : i \in V : \langle \textit{actions} \rangle_i) \end{array}$$

A single action is either a simple assignment or a guarded assignment. A simple assignment can simultaneously assign to several variables. Its meaning is as usual. A guarded assignment may have multiple guards. If more than one guard evaluate to true then one is selected non-deterministically[2]. *Guarded assignments are not allowed to abort or to hang forever if none of their guards evaluate to true and instead they behave like* **skip**.

Additionally, there are the following requirements regarding the well-formedness of a UNITY program: (1) a program has at least one action; (2) actions in a program should only write to the declared write variables and read from the declared read variables; and (3) the set of write variables of a program is included in the set of its read variables. These requirements are so natural that we take them for granted. Yet it must be noted that their precise formulation (not detailed in this paper) is non-trivial and is crucial in proving compositionality laws presented in this paper. See for example [17]. Note that we do not forbid a variable to be declared as a read (write) variable without the program actually ever reading (writing) it.

As an example, Figure 2 displays a (self-stabilizing) UNITY program to compute minimal distance between nodes in a network.

To access each component of a program we introduce the following notation:

PROGRAM COMPONENTS:

$\mathbf{a}P, \mathbf{r}P, \mathbf{w}P$, and $\mathbf{ini}P$ denote respectively the set of all actions, the set of read variables, the set of write variables, and the initial condition that belong to the program P. In addition, $\mathbf{ro}P$ denotes the set of read-only variables of P. $\mathbf{ro}P = \mathbf{r}P - \mathbf{w}P$.

[2] In original UNITY [7] it is required that if multiple guards evaluate to true, then the corresponding assignments must all have the same effect. This requirement is dropped here.

prog MinDist
read $\{a, b : a, b \in V : d.a.b\}$ [a]
write $\{a, b : a, b \in V : d.a.b\}$
init true
assign $(\llbracket a : a \in V : d.a.a := 0)$
$\llbracket (\llbracket a, b : a, b \in V \wedge (a \neq b) : d.a.b := \mathsf{min}\{b' : b' \in E.b : d.a.b' + 1\})$

Figure 2: MinDist in UNITY

[a] Another, more familiar, notation used to denote the above set of variables is: d : **array** V **of array** V **of** Val

4.1 Parallel Composition

Since UNITY actions model parallel components, parallel composition of two UNITY programs amounts to simply 'merging' components of both programs. In UNITY parallel composition is denoted by $\llbracket$. In [7] it is also called *program union*.

Definition 4.1 : Parallel Composition

$$\begin{array}{lcl|lcl} \mathbf{r}(P\llbracket Q) & = & \mathbf{r}P \cup \mathbf{r}Q & \mathbf{w}(P\llbracket Q) & = & \mathbf{w}P \cup \mathbf{w}Q \\ \mathbf{ini}(P\llbracket Q) & = & \mathbf{ini}P \wedge \mathbf{ini}Q & \mathbf{a}(P\llbracket Q) & = & \mathbf{a}P \cup \mathbf{a}Q \end{array}$$

4.2 Primitive Operators

The UNITY logic contains three primitive operators to describe program behavior: unless, ensures, and $\mapsto$. unless describes safety; ensures describes progress achieved through the act of one action; and $\mapsto$ describes progress achieved through coordination of several actions. A special case of unless, namely stability, describes predicates that cannot be destroyed by a program. Progress and stability are required to describe self-stabilization.

In the sequel, P, Q, and R will range over UNITY programs; a, b, and c over actions; and p, q, r, s, J and K over state-predicates.

Definition 4.2 : UNLESS

$$_P\vdash p \text{ unless } q \;=\; (\forall a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{p \vee q\})$$

where $\{p\}\ a\ \{q\}$ denotes a Hoare triple specification with the usual meaning[a].

Definition 4.3 : ENSURES

$$_P\vdash p \text{ ensures } q \;=\; (\ _P\vdash p \text{ unless } q) \;\wedge\; (\exists a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{q\})$$

[a] It does not matter whether it means total or partial correctness since all actions in a UNITY program are assumed to be terminating.

Intuitively, $_P\vdash p$ unless q means that once p holds during an execution of P, it remains to hold at least until q holds. $_P\vdash p$ ensures q encompasses p unless q and additionally there also exists an action that can, and because of the fairness assumption of UNITY, *will* establish q. Notice how the progress to q is guaranteed by one action a. A more general notion of progress is obtained by taking the least transitive and disjunctive closure of ensures. This is the operator $\mapsto$ (read: 'leads-to'). We will not give its formal definition as we will later on introduce a more compositional variant of it.

Here are some examples of properties described using the UNITY primitive operators:

$$_{\mathsf{MinDist}}\vdash \quad \text{true} \;\mapsto\; (\forall a, b : a, b \in V : d.a.b = \delta.a.b) \tag{4.1}$$

$$_{\mathsf{MinDist}}\vdash \quad (\forall a, b : a, b \in V : d.a.b = \delta.a.b) \;\text{ unless }\; \text{false} \tag{4.2}$$

The first property states that the value of all $d.a.b$'s in the program MinDist will eventually be equal to the actual minimal distance from a to b. The second property states that once such a situation is achieved it will remain so forever. Note that together (4.1) and (4.2) implies that MinDist is self-stabilizing. A predicate p satisfying p unless false is called *stable predicate*. Such a predicate is useful to describe self-stabilizing systems and therefore a separate operator is reserved for it:

Definition 4.4 : STABLE PREDICATE

$$_P\vdash \circlearrowleft p \;=\; _P\vdash p \text{ unless false}$$

If both $_P\vdash \circlearrowleft p$ and $[\mathbf{ini}P \Rightarrow p]$ hold then p is an *invariant*. A stable predicate does not have to ever hold during an execution whereas an invariant holds throughout any execution of P. The conjunction and disjunction of two stable predicates are again stable, but weakening or strengthening a stable predicate do not always yield a stable predicate.

Figure 3 displays a number of basic properties of unless, $\circlearrowleft$, and ensures taken from [7]. Theorems analogous to unless INTRODUCTION, POST-WEAKENING, and SIMPLE CONJUNCTION also exist for ensures. There also exist stronger CON-

Theorem 4.5 : unless INTRODUCTION

$$P: \ \frac{[p \Rightarrow q]}{p \text{ unless } q}$$

Theorem 4.6 : POST-WEAKENING

$$P: \ \frac{(p \text{ unless } q) \ \wedge \ [q \Rightarrow r]}{p \text{ unless } r}$$

Theorem 4.7 : SIMPLE CONJUNCTION

$$P: \ \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{(p \wedge r) \text{ unless } (q \vee s)}$$

Theorem 4.8 : SIMPLE DISJUNCTION

$$P: \ \frac{(p \text{ unless } q) \ \wedge \ (r \text{ unless } s)}{(p \vee r) \text{ unless } (q \vee s)}$$

Corollary 4.9 : $\circlearrowleft$ CONJUNCTION

$$P: \ \frac{(\circlearrowleft p) \ \wedge \ (\circlearrowleft q)}{\circlearrowleft (p \wedge q)}$$

Corollary 4.10 : $\circlearrowleft$ DISJUNCTION

$$P: \ \frac{(\circlearrowleft p) \ \wedge \ (\circlearrowleft q)}{\circlearrowleft (p \vee q)}$$

Theorem 4.11 : ensures PROGRESS SAFETY PROGRESS (PSP)

$$P: \ \frac{(p \text{ ensures } q) \ \wedge \ (r \text{ unless } s)}{p \wedge r \text{ ensures } (q \wedge r) \vee s}$$

Figure 3: Some basic laws for unless , $\circlearrowleft$, and ensures

JUNCTION and DISJUNCTION theorems for unless. See [7]. Corollaries 4.9 and 4.10 follow from Theorems 4.7 and 4.8.

Notational convention: if it is clear from the context which program P is meant, it is often omitted from formulas. For example we write p unless q to mean ${}_P\vdash p$ unless q. For laws we write for example:

$$P: \ \frac{\ldots (p \text{ unless } q) \ldots}{r \text{ unless } s} \text{ to abbreviate: } \frac{\ldots \ {}_P\vdash p \text{ unless } q \ldots}{{}_P\vdash r \text{ unless } s}$$

4.3 Compositionality of Safety and Single-action Progress

Compositionality, as explained in Section 2, is a property (of a programming logic) which enables us to split a specification of a composite program into the specifications of its components. It is usually expressed as a law in the form of (2.1). The usefulness of such a property has been motivated in Section 2.

The compositionality of safety properties follows a simple principle: the safety of a program follows from the safety of its components. The compositionality of ensures is also simple: to ensure some progress in a program, it suffices to ensure it by a component program. The other components only need to maintain the safety part of the ensured progress.

Theorem 4.12 : unless COMPOSITIONALITY

$$({}_P\vdash p \text{ unless } q) \ \wedge \ ({}_Q\vdash p \text{ unless } q) \quad = \quad ({}_{P\|Q}\vdash p \text{ unless } q)$$

Corollary 4.13 : $\circlearrowleft$ COMPOSITIONALITY

Follows from Theorem 4.12.

$$({}_P\vdash \circlearrowleft p) \ \wedge \ ({}_Q\vdash \circlearrowleft p) \quad = \quad ({}_{P|Q}\vdash \circlearrowleft p)$$

Theorem 4.14 : ensures COMPOSITIONALITY

$$\frac{({}_P\vdash p \text{ ensures } q) \ \wedge \ ({}_Q\vdash p \text{ unless } q)}{{}_{P\|Q}\vdash p \text{ ensures } q}$$

4.4 General Progress

Unfortunately, compositionality does not directly extend from ensures to the general progress operator $\mapsto$. The definition of $\mapsto$ [7] is simple and elegant, effectively capturing our intuitive notion of progress. But it is too general that no 'strong' compositionality result can be expected. There is also another problem. Under certain circumstances it can be argued that certain progress properties can be preserved by parallel composition [22]. So, general progress is compositional, only not as compositional as ensures . But $\mapsto$ is discovered to carry too few information to be compositional even under this 'weaker' sense [20]. The problem is that $\mapsto$ as defined in [7] does not describe the set of variables upon which the described progress depends on, and this turns out to be crucial for deriving compositionality. For this reason we introduce a variant of $\mapsto$.

In Section 2 it has been sketched how a progress operator can be extended such that it composes nicely with respect to write-disjoint composition (2.2). To refresh the reader's memory let $J \ {}_P\vdash \ p \rightarrowtail q$ describe progress $J \wedge p \mapsto q$. If there are changes made during the progress it will be made on the write variables, so we can as well require that p and q are confined by $\mathbf{w}P$ whereas whatever assumed values of read-only variables should be captured in J. Since the values of read-only variables do not change, obviously J has to be a stable predicate. For soundness reason it is also required that all intermediate predicates from which the progress $p \rightarrowtail q$ is built must also be confined by $\mathbf{w}P$. The new progress operator is called *reach*, denoted by $\rightarrowtail$[3]:

[3]The reader may notice that $\rightarrowtail$ resembles the subscripted $\mapsto$ operator by Sanders [21]. There are two principle differences. Firstly, the 'subscript' J, of $\rightarrowtail$ needs only to be stable whereas Sanders requires it to be an invariant. In the context of program composition, this makes more sense: the environment of a program may cause the program to spring from one stability region to another, which is expressible using different stable J-parameters, but not if J has to be an invariant. Secondly, Sanders does not require confinement by $\mathbf{w}P$, which is crucial to derive the compositionality laws presented later.

Definition 4.15 : REACH

For all P, J, the relation $(\lambda\ p, q.\ J\ {}_P\vdash\ p \rightarrowtail q)$ is defined as the least $\rightarrow$ satisfying:

1. **Ensures lifting**

$$\frac{p, q \in \mathsf{Pred.}(\mathbf{w}P)\ \wedge\ (\circlearrowleft J)\ \wedge\ ({}_P\vdash J \wedge p\ \mathsf{ensures}\ q)}{p \rightarrow q}$$

2. **Transitivity**

$$\frac{p \rightarrow q\ ,\ q \rightarrow r}{p \rightarrow r}$$

3. **Left disjunctivity**
   If we have $p \rightarrow q$ for all $p \in W$ for some non-empty W, then

   $$(\exists p : p \in W : p) \rightarrow q$$

   also holds.

One can easily prove that $\rightarrowtail$ itself also satisfies the above three requirements. Since $\rightarrowtail$ is the least relation satisfying the three properties above, statements of the form $(p \rightarrowtail q) \Rightarrow F(p, q)$ can be proven by showing that the relation F also satisfies the three properties above (replace $\rightarrowtail$ by F). This is called $\rightarrowtail$ *Induction Principle* (in analogy to '$\mapsto$ induction' in [7]). Using the induction one can, for example, easily show that $J\ {}_P\vdash\ p \rightarrowtail q$ implies that p and q are both confined by $\mathbf{w}P$ and that J is stable in P.

The progress (expressed in terms of $\mapsto$) described by $J\ {}_P\vdash\ p \rightarrowtail q$ is ${}_P\vdash\ J \wedge p \mapsto q$. However, $J\ {}_P\vdash\ p \rightarrowtail q$ is not generally equal to $p, q \in \mathsf{Pred.}(\mathbf{w}P) \wedge ({}_P\vdash \circlearrowleft J) \wedge ({}_P\vdash J \wedge p \mapsto q)$. This is caused by the fact that $\rightarrowtail$ is *not* disjunctive in its J-argument, something which also has a subtle consequence on $\rightarrowtail$'s behavior with respect parallel composition. An example later in Section 6 will demonstrate this point.

As an example, consider a program buffer with $\mathbf{w}(\mathsf{buffer}) = \{\mathsf{out}\}$ and $\mathbf{ro}(\mathsf{buffer}) = \{\mathsf{in}\}$. The formula $(\forall X :: (\mathsf{in} = X)\ {}_{\mathsf{buffer}}\vdash\ \mathsf{true} \rightarrowtail (\mathsf{out} = X))$ states that the program buffer will eventually copy the value of in to out. However, $\mathsf{true}\ {}_{\mathsf{buffer}}\vdash\ (\mathsf{in} = X) \rightarrowtail (\mathsf{out} = X)$ is *not* a valid expression because the argument "$\mathsf{in} = X$" is not a predicate confined by $\mathbf{w}(\mathsf{buffer})$.

Figures 4 displays some basic properties of $\rightarrowtail$ which are analogous to those of $\mapsto$. The proofs are also analogous to those found in [7]. Figures 5 displays the properties of $\rightarrowtail$ which have no analogous $\mapsto$ properties. Note that just as in [21] we also have the $\rightarrowtail$ SUBSTITUTION law for free. Note also that although we can strengthen the J in $J\ {}_P\vdash\ p \rightarrowtail q$ with another stable predicate (Theorem STABLE STRENGTHENING) but we cannot generally weaken J.

Corollary 4.16 : ($\Rightarrow$,$\rightarrowtail$) Introduction

Follows from ensures Introduction (analogous to Theorem 4.5) and Definition 4.15 of $\rightarrowtail$

$$P, J : \frac{p, q \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ [J \wedge p \Rightarrow q] \ \wedge \ (\circlearrowleft J)}{p \rightarrowtail q}$$

Theorem 4.17 : $\rightarrowtail$ General Disjunction
For all *finite* and *non-empty* sets W:

$$P, J : \frac{(\forall i : i \in W : p.i \rightarrowtail q.i)}{(\exists i : i \in W : p.i) \rightarrowtail (\exists i : i \in W : q.i)}$$

Theorem 4.18 : Progress Safety Progress (PSP)

$$P, J : \frac{r, s \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (r \wedge J \ \mathsf{unless} \ s) \wedge \ (p \rightarrowtail q)}{(p \wedge r) \rightarrowtail (q \wedge r) \vee s}$$

Theorem 4.19 : Completion
For all *finite* and *non-empty* sets W:

$$P, J : \frac{r \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\forall i : i \in W : q.i \wedge J \ \mathsf{unless} \ r) \ \wedge \ (\forall i : i \in W : p.i \rightarrowtail q.i \vee r)}{(\forall i : i \in W : p.i) \rightarrowtail (\forall i : i \in W : q.i) \vee r}$$

Figure 4: Properties of $\rightarrowtail$ which are analogous to those of $\mapsto$

Theorem 4.20 : $\rightarrowtail$ Stable Shift

$$P : \frac{r \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\circlearrowleft J) \ \wedge \ (J \wedge r \vdash p \rightarrowtail q)}{J \vdash p \wedge r \rightarrowtail q}$$

Theorem 4.21 : $\rightarrowtail$ Stable Strengthening

$$P : \frac{(\circlearrowleft K) \ \wedge \ (J \vdash p \rightarrowtail q)}{J \wedge K \vdash p \rightarrowtail q}$$

Corollary 4.22 : $\rightarrowtail$ Stable Background

$$P : \frac{J \vdash p \rightarrowtail q}{\circlearrowleft J}$$

Theorem 4.23 : $\rightarrowtail$ Confinement

$$P, J : \frac{p \rightarrowtail q}{p, q \in \mathsf{Pred.}(\mathbf{w}P)}$$

Theorem 4.24 : $\rightarrowtail$ Substitution

$$P, J : \frac{\begin{array}{c}[J \wedge p \Rightarrow q] \ \wedge \ [J \wedge r \Rightarrow s] \\ p, s \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (q \rightarrowtail r)\end{array}}{p \rightarrowtail s}$$

Figure 5: More properties of $\rightarrowtail$

Now how about the compositionality of $\rightarrowtail$? After-all, this was the reason why we introduced it. Well, $\rightarrowtail$ satisfies Singh kind of compositionality [22] and additionally also some stronger persistence principle (2.2), but more elaboration will be delayed until Section 6. First we would like to discuss how the notion of self-stabilization can be formally expressed and what its laws are.

5 Stabilization

A self-stabilizing system is assumed to work in an unstable environment (often called '*adversary*') which may produce transient errors or undergo a reconfiguration ('*sabotage*' by the adversary), each of which affects the consistency of the variables upon which the self-stabilization depends. We do not have to model an adversary explicitly by a process. Instead, we can insist that any sabotage will bring the system to some 'failure' state. A system which can stabilize from initial condition J to some condition q will be able to 're-stabilize' if its adversary pushes it to some failure state in J. So, the initial condition of a stabilizing system models the set of possible failure states to which the system will fall after sabotages. To a great extend, this models the behavior of an adversary. The above observation implies that stabilization in program P can be treated in isolation (we can treat it as a property of P rather than a property of $P[\!]\mathsf{adversary}$).

In the case the adversary can do anything, then we must assume the initial condition true, which corresponds to the classical notion of self-stabilization. In Linear Temporal Logic *"P self-stabilizes to q"* can be expressed by $P \vdash \Diamond\Box q$. In UNITY this can be expressed by:

$$(\exists q' :: (\mathsf{true}\ {}_P\!\vdash \mathsf{true} \rightarrowtail q' \wedge q)\ \wedge\ ({}_P\!\vdash \circlearrowleft (q' \wedge q))) \tag{5.1}$$

The existential quantification may seem strange at first, but notice that in $P \vdash \Diamond\Box q$ the situation $\Box q$ does not have to hold immediately after the first time q holds, but perhaps only after several iterations. The predicate q' basically encodes the end of such an iteration (after which q will continue to hold).

A more general setup where the adversary is not totally uncooperative (or if we only want to guarantee correctness under a less fatalistic assumption) amounts to replacing the two true's in (5.1) by parameters, say, J and p. This is in fact is the definition of *convergence*, introduced by Arora and Gouda in [5]. Closely related concepts are the concepts of *adaptiveness* as in [11] and *leads-to-stabilization* as in [15]. Convergence turns out to enjoy many interesting properties. Its formal definition is below.

Definition 5.1 : CONVERGENCE

$$J\ {}_P\!\vdash p \rightsquigarrow q$$
$$=$$
$$q \in \mathsf{Pred.}(\mathbf{w}P)\ \wedge\ (\exists q' :: ({}_P\!\vdash \circlearrowleft (J \wedge q' \wedge q)) \wedge (J\ {}_P\!\vdash p \rightarrowtail q' \wedge q))$$

So, convergence is not self-stabilization. It is more general. It is the ability

to stabilize to certain conditions from certain pre-conditions. Note also that $p \leadsto q$ does not imply that q is a stable predicate. It only states that eventually q will hold forever.

Figure 6 displays a number of basic properties of convergence. Notice that $\leadsto$ is, in contrast to $\rightarrowtail$, not only $\vee$-junctive, but also $\wedge$-junctive. This makes $\leadsto$ calculationally attractive. The next section will present some useful compositionality properties of both operators.

6 Compositionality

Subsection 4.3 shows that safety (unless) and single action progress (ensures) are highly compositional. Now we will look at the compositionality of general progress and convergence. Note that it is crucial that progress (and the progress part of convergence) is modelled by $\rightarrowtail$ instead of the traditional $\mapsto$. If the latter is used, no interesting compositionality result can be derived.

A general compositionality property of progress is proposed by the Singh [22]. Consider two programs, P and Q. If we execute P and Q in parallel, then basically Q can destroy any progress $p \rightarrowtail q$ in P by writing to a shared variable of P and Q. However, if we know that under condition r, Q will always announce any modification to the shared variables by establishing s, then starting from $p \wedge r$ the program $P[\!]Q$ will either reach q (through the actions of P), or Q writes to some shared variables and spoils the progress, but in this case we know that s will hold. This is an instance of the Singh 'Law'[4]. Before we give the formulation of the law, we will introduce some definitions.

Definition 6.1 : !?

$$P?!Q \;=\; \mathbf{r}P \cap \mathbf{w}Q$$

Definition 6.2 : unless_V

$$_Q\vdash p \;\mathsf{unless}_V\; q \;=\; (\forall s ::\; _Q\vdash p \wedge (\forall v : v \in V : v = s.v)\; \mathsf{unless}\; q)$$

Note the dummy s ranges over states ($\mathsf{Var} \rightarrow \mathsf{Val}$). Alternatively, by omitting some overloading the above is equal to:

$$_Q\vdash p \;\mathsf{unless}_V\; q \;=\; (\forall s ::\; _Q\vdash p \wedge (\lambda t.\ (\forall v : v \in V : t.v = s.v))\; \mathsf{unless}\; q)$$

So, $P?!Q$ is the set of variables through which P reads updates made by Q. If P and Q communicate through channels, then $P?!Q$ are the channels from Q to P. $_Q\vdash p \;\mathsf{unless}_V\; q$ means that under condition p, every time Q modifies any variable in V, it will mark the event by establishing q. We are particularly

[4]When we first tried to replay Singh's proof of the 'Law' using Theorem Prover HOL, we quickly discovered that the proof is flawed. While Singh 'Law' predicts the kind of compositionality one naturally expects for progress properties, the law is simply underivable when progress is modelled with $\mapsto$. That is why we introduced $\rightarrowtail$. Using $\rightarrowtail$ we were able to replay Singh's proof and hence derived the Law.

Corollary 5.2 : ($\leadsto$,$\mapsto$) Conversion

$$P, J : \frac{p \leadsto q}{p \mapsto q}$$

Follows from the definition of $\leadsto$ and Theorem 4.24.

Corollary 5.3 : $\leadsto$ Stable Background

$$P : \frac{J \vdash p \leadsto q}{\circlearrowright J}$$

Follows from Corollaries 5.2 and 4.22.

Corollary 5.4 : $\leadsto$ Confinement

$$P, J : \frac{p \leadsto q}{p, q \in \mathsf{Pred.}(\mathbf{w}P)}$$

Follows from Corollary 5.2 and Theorem 4.23.

Corollary 5.5 : (ensures,$\leadsto$) Introduction

$$P, J : \frac{p, q \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\circlearrowright J) \ \wedge \ (\circlearrowright (J \wedge q)) \ \wedge \ (p \wedge J \ \mathsf{ensures} \ q)}{p \leadsto q}$$

Corollary 5.6 : ($\Rightarrow$,$\leadsto$) Introduction

$$P, J : \frac{[p \wedge J \Rightarrow q] \ \wedge \ p, q \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\circlearrowright J) \ \wedge \ (\circlearrowright (J \wedge p))}{p \leadsto q}$$

Corollary 5.7 : $\leadsto$ Substitution

$$P, J : \frac{[J \wedge p \Rightarrow q] \ \wedge \ [J \wedge r \Rightarrow s] \ \wedge \ p, s \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (q \leadsto r)}{p \leadsto s}$$

Theorem 5.8 : Accumulation

$$P, J : \frac{(p \leadsto q) \ \wedge \ (q \leadsto r)}{p \leadsto q \wedge r}$$

Theorem 5.9 : $\leadsto$ Transitivity

$$P, J : \frac{(p \leadsto q) \ \wedge \ (q \leadsto r)}{p \leadsto r}$$

Follows from Theorem 5.8 and Corollaries 5.4 and 5.4.

Theorem 5.10 : Disjunction

$$P, J : \frac{(p \leadsto q) \ \wedge \ (r \leadsto s)}{p \vee r \leadsto q \vee s}$$

Theorem 5.11 : $\leadsto$ Conjunction
For all *non-empty* and *finite* sets W:

$$P, J : \frac{(\forall i : i \in W : p.i \leadsto q.i)}{(\forall i : i \in W : p.i) \leadsto (\forall i : i \in W : q.i)}$$

Theorem 5.12 : $\leadsto$ Stable Shift

$$P : \frac{p' \in \mathsf{Pred.}\mathbf{w}P \wedge (\circlearrowright J) \ \wedge \ (J \wedge p' \vdash p \leadsto q)}{J \vdash p \wedge p' \leadsto q}$$

Figure 6: Some basic properties of $\leadsto$

interested in the case of $V = P?!Q$. For example, ${}_Q\vdash \mathsf{true}\ \mathsf{unless}_{P?!Q}\ q$ states that Q cannot disturb P without 'raising the flag' q, and ${}_Q\vdash p\ \mathsf{unless}_{P?!Q}\ \mathsf{false}$ states that Q cannot disturb P while p holds.

The Singh Law is given below. The proof [18] is analogous to the (erroneous) proof for the $\mapsto$ version of the Law given in [22]. Some interesting corollaries of the Law is also given below.

Theorem 6.3 : SINGH LAW

$$\frac{r, s \in \mathsf{Pred.}\mathbf{w}(P[\!]Q)\ \wedge\ p_1 \in \mathsf{Pred.}(\mathbf{w}P \cup (P?!Q)) \quad (\ {}_{P[\!]Q}\vdash \circlearrowright J)\ \wedge\ (\ {}_Q\vdash r \wedge J\ \mathsf{unless}_{P?!Q}\ s)\ \wedge\ (J \wedge p_1\ {}_P\vdash p_2 \rightarrowtail q)}{J\ {}_{P[\!]Q}\vdash p_1 \wedge p_2 \wedge r \rightarrowtail q \vee \neg p_1 \vee \neg r \vee s}$$

Corollary 6.4 : until COMPOSITIONALITY 1

$$\frac{(\ {}_Q\vdash \circlearrowright J)\ \wedge\ (\ {}_Q\vdash J \wedge p\ \mathsf{unless}_{P?!Q}\ q) \quad (\ {}_P\vdash J \wedge p\ \mathsf{unless}\ q)\ \wedge\ (J\ {}_P\vdash p \rightarrowtail q)}{J\ {}_{P[\!]Q}\vdash p \rightarrowtail q}$$

Corollary 6.5 : until COMPOSITIONALITY 2

$$\frac{p \in \mathsf{Pred.}(\mathbf{w}P \cup (P?!Q))\ \wedge\ (\ {}_{P[\!]Q}\vdash \circlearrowright J) \quad (\ {}_Q\vdash J \wedge p\ \mathsf{unless}_{P?!Q}\ q)\ \wedge\ (J \wedge p\ {}_P\vdash \mathsf{true} \rightarrowtail q)}{J\ {}_{P[\!]Q}\vdash p \rightarrowtail q}$$

Both Corollaries state sufficient conditions for a progress property to be preserved by the parallel composition. They express the compositionality of a more restricted progress property which in linear temporal logic is called '*until*' $((J \wedge p\ \mathsf{unless}\ q) \wedge (J \vdash p \rightarrowtail q)$ corresponds with "$J \wedge p\ \mathsf{until}\ q$" in linear temporal logic). The corollaries derive easily and nicely from the Law. Let us share this with the reader by presenting the proof of Corollary 6.5:

Proof:

$$\begin{array}{ll} & J\ {}_{P[\!]Q}\vdash p \rightarrowtail q \\ \Leftarrow & \quad \{ \rightarrowtail \text{SUBSTITUTION} \} \\ & (J\ {}_{P[\!]Q}\vdash p \rightarrowtail (p \wedge q) \vee q)\ \wedge\ q \in \mathsf{Pred.}(\mathbf{w}(P[\!]Q)) \\ \Leftarrow & \quad \{ \rightarrowtail \text{PSP} \} \\ & (J\ {}_{P[\!]Q}\vdash p \rightarrowtail q \vee \neg p)\ \wedge\ (\ {}_{P[\!]Q}\vdash J \wedge p\ \mathsf{unless}\ q)\ \wedge\ q \in \mathsf{Pred.}(\mathbf{w}(P[\!]Q)) \end{array}$$

$q \in \mathsf{Pred.}(\mathbf{w}(P[\!]Q))$ follows from ${}_P\vdash \mathsf{true} \rightarrowtail q$ and $\rightarrowtail$ CONFINEMENT. The progress part follows directly from the assumptions by the application of the SINGH LAW by instantiating p_1, p_2, q, r, s, J with $p, \mathsf{true}, q, p, q, j$.

As for the unless part:

$$\begin{array}{ll} & {}_{P[\!]Q}\vdash J \wedge p\ \mathsf{unless}\ q \\ \Leftarrow & \quad \{ \mathsf{unless}\ \text{COMPOSITIONALITY} \} \\ & (\ {}_P\vdash J \wedge p\ \mathsf{unless}\ q)\ \wedge\ (\ {}_Q\vdash J \wedge p\ \mathsf{unless}\ q) \\ \Leftarrow & \quad \{ \mathsf{unless}\ \text{POST-WEAKENING; Definition}\ \circlearrowright \} \end{array}$$

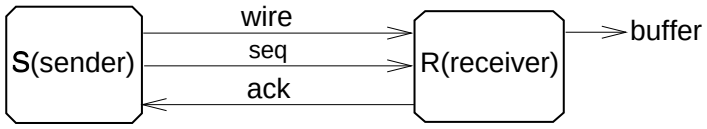

Figure 7: A simple protocol.

$$
\begin{array}{ll}
 & ({}_{P}\vdash \circlearrowleft(J \wedge p)) \ \wedge \ ({}_{Q}\vdash J \wedge p \ \mathsf{unless}\ q) \\
\Leftarrow & \quad \{\ \mathsf{unless}\ \textsc{Simple Disjunction};\ \text{Definition}\ \ \mathsf{unless}_{V}\ \ \} \\
 & ({}_{P}\vdash \circlearrowleft(J \wedge p)) \ \wedge \ ({}_{Q}\vdash J \wedge p \ \mathsf{unless}_{P?!Q}\ q) \\
\Leftarrow & \quad \{\ \rightarrowtail\ \textsc{Stable Background}\ \} \\
 & (J \wedge p \ {}_{P}\vdash \mathsf{true} \rightarrowtail q) \ \wedge \ ({}_{Q}\vdash J \wedge p \ \mathsf{unless}_{P?!Q}\ q)
\end{array}
$$

■

As an example, let us consider a simple protocol as displayed in Figure 7. The task of the protocol is to establish the progress (S is the sender and R is the receiver):

$$(\forall X :: J \ {}_{R[\!]S}\vdash (\mathsf{wire} = X) \rightarrowtail (\mathsf{buffer} = X))$$

for some invariant J. The sender tags each new message it puts on the wire by some sequence number. The receiver acknowledges a message by returning its sequence number to the sender. So, an acknowledged message can be identified by $\mathsf{seq} = \mathsf{ack}$. The sender can be modelled by the following program:

$$S\text{:}\quad \text{if } \mathsf{seq}{=}\mathsf{ack}\ \textbf{then}\ \mathsf{wire}, \mathsf{seq} := \textit{produce new message}, \mathsf{seq}^{+}$$

Where n^{+} produces a number, different from n. So, S can only put something new on the wire if the current message on the wire is already acknowledged. Consequently, for the system to make progress, any message sent must eventually be acknowledged. This can be expressed as follows:

$$J \ {}_{R[\!]S}\vdash \mathsf{true} \rightarrowtail (\mathsf{seq} = \mathsf{ack})$$

It seems reasonable to assign the task above to the receiver. Let us see now how this is formally justified using the Singh Law:

$$
\begin{array}{ll}
 & J \ {}_{R[\!]S}\vdash \mathsf{true} \rightarrowtail (\mathsf{seq} = \mathsf{ack}) \\
\Leftarrow & \quad \{\ \rightarrowtail\ \textsc{Disjunction}\ (\text{Theorem } 4.17)\ \} \\
 & (\forall X :: J \ {}_{R[\!]S}\vdash (\mathsf{seq} = X) \rightarrowtail (\mathsf{seq} = \mathsf{ack})) \\
\Leftarrow & \quad \{\ \text{Corollary } 6.5\ \} \\
 & (\forall X :: \begin{array}{l} ({}_{S}\vdash (J \wedge \mathsf{seq} = X) \ \mathsf{unless}_{R?!S}\ (\mathsf{seq} = \mathsf{ack})) \wedge \\ (J \wedge (\mathsf{seq} = X) \ {}_{R}\vdash \mathsf{true} \rightarrowtail (\mathsf{seq} = \mathsf{ack}))\ ) \end{array}
\end{array}
$$

Note that the resulting progress specification states that making ack equal to seq is now R's responsibility. One can also prove that the resulting unless specification can be strengthened to ${}_S\vdash$ (seq $\neq$ ack) unless$_{R?!S}$ false, which states that S cannot send anything new as long as seq $\neq$ ack.

The Singh Law describes how two arbitrary parallel programs can influence each other's progress. It is a very general law. In many cases however, we know more about how the component programs interact through their shared variables. That knowledge can be exploited to derive more constructive compositionality properties.

6.1 Write-Disjoint Composition

A much nicer compositionality rule can be obtained for parallel composition of write-disjoint programs, that is, programs which share no common write variable. "P and Q are write-disjoint programs" is written $P \div Q$.

Definition 6.6 : Write-disjoint Programs

$$P \div Q \quad = \quad (\mathbf{w}P \cap \mathbf{w}Q = \emptyset)$$

In a network of write-disjoint programs, each program can only write to its own variables or to other program's read-only variables. Consequently, if P and Q are write-disjoint and $p \in \mathsf{Pred.}(\mathbf{w}P)$ then Q cannot destroy p:

$$\frac{(P \div Q) \ \wedge \ p \in \mathsf{Pred.}(\mathbf{w}P)}{{}_Q\vdash \circlearrowleft p} \tag{6.1}$$

This is a crucial property in deriving the Transparency principle (2.2) we mentioned in Section 2.

We have defined $\rightarrowtail$ in such a way that all intermediate pairs $p' \rightarrowtail q'$ required to construct $J \ {}_P\vdash \ p \rightarrowtail q$ (via Transitivity or Left Disjunctivity) are confined by $\mathbf{w}P$. Consequently, by (6.1), if we have another program Q which is write-disjoint with P and which also respects $\circlearrowleft J$ then Q cannot destroy any of those intermediate progress properties and hence $p \rightarrowtail q$ is also constructible in $P[\!]Q$. So, $\rightarrowtail$ satisfies the Transparency principle.

Theorem 6.7 : $\rightarrowtail$ Transparency

$$\frac{P \div Q \ \wedge \ ({}_Q\vdash \circlearrowleft J) \ \wedge \ (J \ {}_P\vdash \ p \rightarrowtail q)}{J \ {}_{P[\!]Q}\vdash \ p \rightarrowtail q}$$

The theorem can be easily derived from the fact that $\rightarrowtail$ is defined as the *least* relation satisfying transitivity, disjunctivity, and ensures-lifting as described in Definition 4.15 of $\rightarrowtail$. So, it suffices to show that the relation $(\lambda p, q.\ J \ {}_{P[\!]Q}\vdash \ p \rightarrowtail q)$ in the conclusion (of above theorem) satisfies the same properties.

An analogous law also holds for convergence.

Theorem 6.8 : $\leadsto$ TRANSPARENCY

$$\frac{P \div Q \ \wedge \ (_{Q}\vdash \circlearrowleft J) \ \wedge \ (J \ _{P}\vdash p \leadsto q)}{J \ _{P\|Q}\vdash p \leadsto q}$$

The Transparency principle is fundamental for write-disjoint composition. Some well known design techniques that we use in practice are corollaries of this principle. A progress or convergence property is usually constructed, either using transitivity, disjunction, or conjunction principles, from a number of simpler progress/convergence properties. Using the principle we can delegate each constituent property, if we so desire, to be realized by a write-disjoint component of a program. This is formulated by the following theorem.

Theorem 6.9 : SPIRAL LAW

$$\frac{(P \div Q) \ \wedge \ (_{P}\vdash \circlearrowleft (J \wedge s)) \ \wedge \ (_{Q}\vdash \circlearrowleft J) \quad (J \ _{P}\vdash p \rightarrowtail s) \ \wedge \ (J \wedge s \ _{Q}\vdash \mathsf{true} \rightarrowtail r)}{J \ _{P\|Q}\vdash p \rightarrowtail s \wedge r}$$

The proof is nice to look at: **Proof:**

$J \ _{P\|Q}\vdash p \rightarrowtail s \wedge r$

$\Leftarrow$ { $\rightarrowtail$ TRANSITIVITY and PSP }

$(J \ _{P\|Q}\vdash p \rightarrowtail s) \ \wedge \ (J \ _{P\|Q}\vdash s \rightarrowtail r) \ \wedge \ (_{P\|Q}\vdash \circlearrowleft (J \wedge s))$

$\Leftarrow$ { $\rightarrowtail$ STABLE SHIFT, STABLE BACKGROUND, and CONFINEMENT }

$(J \ _{P\|Q}\vdash p \rightarrowtail s) \ \wedge \ (J \wedge s \ _{P\|Q}\vdash \mathsf{true} \rightarrowtail r) \ \wedge \ (_{P\|Q}\vdash \circlearrowleft (J \wedge s))$

$\Leftarrow$ { $\rightarrowtail$ TRANSPARENCY; $\circlearrowleft$ COMPOSITIONALITY; assumptions }

$(J \ _{P}\vdash p \rightarrowtail s) \ \wedge \ (J \wedge s \ _{Q}\vdash \mathsf{true} \rightarrowtail r) \ \wedge \ (_{Q}\vdash \circlearrowleft (J \wedge s))$

$\Leftarrow$ { (6.1) and $\rightarrowtail$ STABLE BACKGROUND }

$(J \ _{P}\vdash p \rightarrowtail s) \ \wedge \ (J \wedge s \ _{Q}\vdash \mathsf{true} \rightarrowtail r) \ \wedge \ (_{Q}\vdash \circlearrowleft J) \ \wedge \ s \in \mathsf{Pred}.(\mathbf{w}P)$

$\Leftarrow$ { $\rightarrowtail$ CONFINEMENT }

$(J \ _{P}\vdash p \rightarrowtail s) \ \wedge \ (J \wedge s \ _{Q}\vdash \mathsf{true} \rightarrowtail r) \ \wedge \ (_{Q}\vdash \circlearrowleft J)$

■

The Spiral law is used to implement a sequential division of tasks. For example if we want to do a broadcast, we can think of a two-steps process: first, construct a spanning tree, and then do the actual broadcast. Usually we have separate programs for both tasks. The Spiral Law provides the required justification for this kind of separation, where in this case P constructs the spanning tree and Q performs the broadcast under the assumption that s describes the existence of this spanning tree. Typically, the law is applied when P and Q form a *layering*. A layering, it has been mentioned in Section 2, is a parallel composition of two write-disjoint programs in which the computation of one program depends on the other (but not necessarily the other way around).

Now recall again the program MinDist (Figure 2) The program computes of the minimal distance between any two vertices in a network. It is known that

the minimal distances *from* a vertex a and the minimal distances *from* a different vertex b can be computed independently. This suggests a parallel division of tasks. The following law fits well in this kind of decomposition. The law is typically applied when P and Q form a *fork* (P and Q do not write to each other, but they share read-only variables) or *non-interfering* (P and Q share no variable) parallel composition.

Theorem 6.10 : CONJUNCTION BY $[\!]$
For any *non-empty* and *finite* set W:

$$J: \frac{(\forall i,j : i,j \in W \wedge (i \neq j) : P.i \div P.j) \quad (\forall i : i \in W : \ {}_{P.i}\vdash p.i \leadsto q.i)}{{}_{([\!]i:i\in W:P.i)}\vdash (\forall i : i \in W : p.i) \leadsto (\forall i : i \in W : q.i)}$$

6.2 A Note on $\rightarrowtail$: Is It Really Different from the Old $\mapsto$ Operator?

When $\rightarrowtail$ is introduced in Subsection 4.4 it is remarked that $\rightarrowtail$ differs essentially from an ordinary progress operator such as $\mapsto$. One may ask if $J \ {}_P\vdash p \rightarrowtail q$ is actually not the same as $({}_P\vdash \circlearrowleft J) \ \wedge \ p,q \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ ({}_P\vdash J \wedge p \mapsto q)$. The implication from left to right does indeed hold, but the one from right to left does not. Consider the following program:

prog P
read $\{a, x\}$
write $\{x\}$
init true
assign if $a = 0$ then $x := 1$ $[\!]$ if $a = 1$ then $x := 1$

In the above program we have $(b = 0) \wedge a < 2 \mapsto (x = 1)$. We assume the reader knows the meaning of $\mapsto$ as in [7]. We expect that the $\rightarrowtail$ version of this property, namely $(b = 0) \wedge a < 2 \vdash \mathsf{true} \rightarrowtail (x = 1)$, also holds. But it does not hold. It cannot hold either, for we will then get an unsound logic. Consider the program TikToe below:

prog TikToe
read $\{a, b\}$
write $\{a\}$
init true
assign if $a = 0$ then $a := 1$ $[\!]$ if $a = 1$ then $a := 0$ $[\!]$ if $b \neq 0$ then $a := a + 1$

The programs P and TikToe are write-disjoint. Suppose $(b = 0) \wedge a < 2 \ {}_P\vdash \mathsf{true} \rightarrowtail (x = 1)$ holds. The predicate $(b = 0) \wedge a < 2$ is also stable in Tiktoe. By the TRANSPARENCY principle we conclude that $(b = 0) \wedge a < 2 \ {}_{P[\!]\mathsf{TikToe}}\vdash \mathsf{true} \rightarrowtail (x = 1)$ also holds. But this simply cannot be true. Consider the execution:

[if $a = 0$ then $a := 1$; if $a = 0$ then $x := 1$; if $a = 1$ then $a := 0$;
if $a = 1$ then $x := 1$; if $b \neq 0$ then $a := a + 1$]*

which is a fair execution of $P[\!]\mathsf{TikToe}$, but with this execution x will never be equal to 1 if initially $x \neq 1 \wedge a < 2 \wedge (b = 0)$.

The property $(b = 0) \wedge a < 2 \mapsto (x = 1)$ can be concluded because we have $(b = 0) \wedge (a = 0)$ ensures $(x = 1)$ and $(b = 0) \wedge (a = 1)$ ensures $(x = 1)$ and we can join them using the disjunctivity of $\mapsto$. So obviously the same disjunctive property does not apply to the stable argument (the J) of the $\rightarrowtail$ operator, and this is the trade-off we pay for getting the TRANSPARENCY principle.

7 Bounded Progress

Many self-stabilizing algorithms rely on step-by-step, inductive stabilization strategies to achieve their goal. Consequently their formal reasoning relies heavily on finding a well-founded relation to prove the progress part while preserving stability. The difficulty is of course in finding the right well-founded relation, which in many cases as in [1, 8, 14] is not easy. But this is something that even the most elegant programming logic cannot help us. That is, this problem of finding the right well-founded relation is something intrinsic in the chosen stabilization strategy. Eventually, one will have to 'encode' his inductive strategy in a formal proof. There are various ways to do this encoding, but we believe that effective capturing of the way we deploy our reasoning strategy in the form of laws will greatly improve the effectiveness of our formal reasoning. So, in this section we will give the formulation of an inductive stabilization strategy called *Round Decomposition*. The name is due to Lentfert and Swierstra [15] but they actually use the name to refer to a more specialized strategy than ours. For the sake of completeness we will first give the formulation of general well-founded induction on progress and convergence.

7.1 General Well-founded Induction on Progress and Convergence

A well-founded relation over A is a relation $\prec \in A \times A$ such that it is not possible to form an infinitely decreasing sequence. A well-founded relation satisfies the well-founded induction principle (in fact, they are equivalent).

Theorem 7.1 : WELL-FOUNDED INDUCTION
For any well-founded relation $\prec \in A \to A \to \mathsf{bool}$:

$$(\forall y : y \in A : (\forall x : x \prec y : X.x) \Rightarrow X.y) \;=\; (\forall y : y \in A : X.y)$$

Many well known structures are well-founded. Natural numbers ordered by $<$ and finite directed acyclic graphs (dags) ordered by the transitive closure of the edge relation (proper ancestor relation) are examples thereof.

Suppose we have a function m that maps program states to A and we have a well-founded relation $\prec$ defined on A. Suppose that the program is such that either it decreases the value of m with respect to $\prec$, or it reaches q. From

the well-foundness of $\prec$ it follows that the program cannot decrease m forever and hence q must eventually hold. This principle is well known; we call it here *Bounded Progress* principle and we call m the *bound function*. The principle applies for progress by $\rightarrowtail$ and also for convergence.

Let $\prec$ be a *well founded* relation over a *non-empty* set A and let m be some metric function (also called *bound function*) that maps states of program P to A.

Theorem 7.2 : $\rightarrowtail$ BOUNDED PROGRESS

$$P, J : \frac{q \in \mathsf{Pred.w}P \qquad (\forall M : M \in A : p \wedge (m = M) \rightarrowtail (p \wedge (m \prec M)) \vee q)}{p \rightarrowtail q}$$

Theorem 7.3 : $\rightsquigarrow$ BOUNDED PROGRESS

$$P, J : \frac{(q \rightsquigarrow q) \ \wedge \ (\forall M : M \in A : p \wedge (m = M) \rightsquigarrow (p \wedge (m \prec M)) \vee q)}{p \rightsquigarrow q}$$

Note: with some overloading omitted the expression

$$p \wedge (m = M) \rightsquigarrow (p \wedge (m \prec M)) \vee q$$

can be written as:

$$p \wedge (\lambda s.\ m.s = M) \rightsquigarrow (p \wedge (\lambda s.\ m.s \prec M)) \vee q$$

7.2 Dividing Execution into Rounds

A progress property is constructed from smaller progress properties individually made by the actions. Alternatively, we can impose that the progress go through a finite sequence of rounds. Each round is associated with a certain obligation. For this to work the original progress property has to be decomposable into round-obligations. This scheme is particularly useful when the actions are yet to be invented.

Actually, the idea is the same as in loop decomposition in sequential programming, except that the rounds do not have to be totally ordered. Given a loop specification we break it into an invariant and a loop-guard. A *round* is now one iteration. At the end of the round the loop invariant must be re-established, which becomes our round-wise *obligation*. Showing termination is the same as showing that we have finitely many rounds. When the loop is finished, we know that the loop-guard is false. This, together with whatever achieved during the last round establishes the specification of the loop. The difference with a sequential system is that in a parallel system (some) rounds can be traversed in parallel (because they are no longer totally ordered).

For a certain class of converging systems, stronger round-wise specifications can be obtained. Suppose now we want to establish $(\forall n : n \in A : q.n)$. Our

strategy is simple. We take A as our set of rounds, assuming we can find a well-founded ordering $\prec$ on A we impose that each round n is to *converge* to $q.n$. Obviously, after passing round n, $q.n$ continues to hold. Hence, after all rounds are passed $(\forall n : n \in A : q.n)$ holds. The nice thing is that when trying to establish $q.n$ in round n we actually know more. For all rounds m that precede n in the ordering $\prec$ the corresponding $q.m$ must have been established and continues to hold. Hence, the conjunction of them, namely $(\forall m : m \prec n : q.m)$, also holds and can be exploited to establish $q.n$. If we have a parallel/distributed system, $q.n$ typically expresses what each process in the system must do for round n. Consequently, $(\forall m : m \prec n : q.m)$ contains information of what neighboring processes achieve so far.

The above principle is called *round decomposition*. It is formulated below. Examples in Part II will show how the principle is used in practice.

Theorem 7.4 : Round Decomposition
For any finite and non-empty set A and any well-founded relation $\prec \in A \times A$:

$$P: \quad \frac{(\circlearrowleft J) \ \wedge \ (\forall n : n \in A : J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \leadsto q.n)}{J \vdash \mathsf{true} \leadsto (\forall n : n \in A : q.n)}$$

The proof is simple and nice to see: **Proof:**
(Let P be a UNITY program) we derive:

$$\begin{array}{ll} & J \vdash \mathsf{true} \leadsto (\forall n : n \in A : q.n) \\ \Leftarrow & \quad \{ \leadsto \text{Conjunction} \} \\ & (\forall n : n \in A : J \vdash \mathsf{true} \leadsto q.n \\ \Leftarrow & \quad \{ \text{Well-founded Induction} \} \\ & (\forall n : n \in A : (\forall m : m \prec n : J \vdash \mathsf{true} \leadsto q.m) \ \Rightarrow \ (J \vdash \mathsf{true} \leadsto q.n)) \end{array}$$

If n is a minimal element then:

$$\begin{array}{ll} & (\forall m : m \prec n : J \vdash \mathsf{true} \leadsto q.m) \ \Rightarrow \ (J \vdash \mathsf{true} \leadsto q.n) \\ \Leftarrow & \quad \{ \text{predicate calculus} \} \\ & J \vdash \mathsf{true} \leadsto q.n \\ = & \quad \{ n \text{ is a minimal element, hence there is no } m \text{ such that } m \prec n \} \\ & J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \leadsto q.n \end{array}$$

If n is not a minimal element then:

$$\begin{array}{ll} & (\forall m : m \prec n : J \vdash \mathsf{true} \leadsto q.m) \ \Rightarrow \ (J \vdash \mathsf{true} \leadsto q.n) \\ \Leftarrow & \quad \{ \leadsto \text{Conjunction} \} \\ & (J \vdash \mathsf{true} \leadsto (\forall m : m \prec n : q.m)) \ \Rightarrow \ (J \vdash \mathsf{true} \leadsto q.n) \\ \Leftarrow & \quad \{ \leadsto \text{Transitivity} \} \\ & J \vdash (\forall m : m \prec n : q.m) \leadsto q.n \\ \Leftarrow & \quad \{ \leadsto \text{Stable Shift} \} \end{array}$$

$$
\begin{array}{l}
\quad (\forall m : m \prec n : q.m) \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\circlearrowleft J) \\
\quad \wedge \\
\quad (J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \leadsto q.n) \\
\Leftarrow \quad \{ \leadsto \textsc{Confinement} \ ; \text{confinement is preserved by } \forall \ \} \\
\quad (\circlearrowleft J) \ \wedge \ (J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \leadsto q.n)
\end{array}
$$

■

8 Conclusion

We have introduced an extension of UNITY which will enable us to formally reason about self-stabilizing systems. The operators and the notational style introduced are concise and carry enough detail to allow a nice set of compositionality laws. We seldom see a really formal proof of distributed programs, and those that attempt, such as in [15], tend to end up with overly complicated proofs. We believe that the situation can be improved if one can learn to translate, in a natural way, novel but intuitive ideas to the formal level. Part II presents three examples of increasing complexity, demonstrating how informal reasoning strategies can be effectively translated to formal proofs using various laws introduced in this paper.

All displayed theorems and corollaries have also been mechanically verified with a general purpose, extensible theorem prover HOL, which is based on Higher Order Logic. All our mechanized theorems are available in the form of libraries and therefore can be re-used to mechanically verify user programs. For example, they have been used to mechanically verify the program MinDist in Figure 2 as well as a hierarchical version thereof (many actual networks are hierarchically organized, and therefore hierarchical self-stabilization makes an interesting application).

Figure 8 displays the general structure of our mechanical verification work with HOL. The core of this work is the UNITY module, which includes the standard UNITY, Sander's extension [21], and the extension presented in this paper. A simple language to construct actions has also been added. So far, the module has been applied to verify a simple alternating bit protocol, a generalized version of the program MinDist, and a hierarchical version thereof. Verifying the alternating bit protocol was very simple. The verification of the generalized version of MinDist is not, and also requires a lot of knowledge on general mathematics, such as lattice theory, graph theory, theory on well-founded relation, and so on. HOL has to be extended with these theories before we can use them. At the time we wrote our modules, only theories on sets are available in HOL. The most work in the verification of MinDist is *not* in the application of the programming logic but rather, in the application of theories such as lattice theory, graph theory, and so on, which are not directly related to the programming logic:

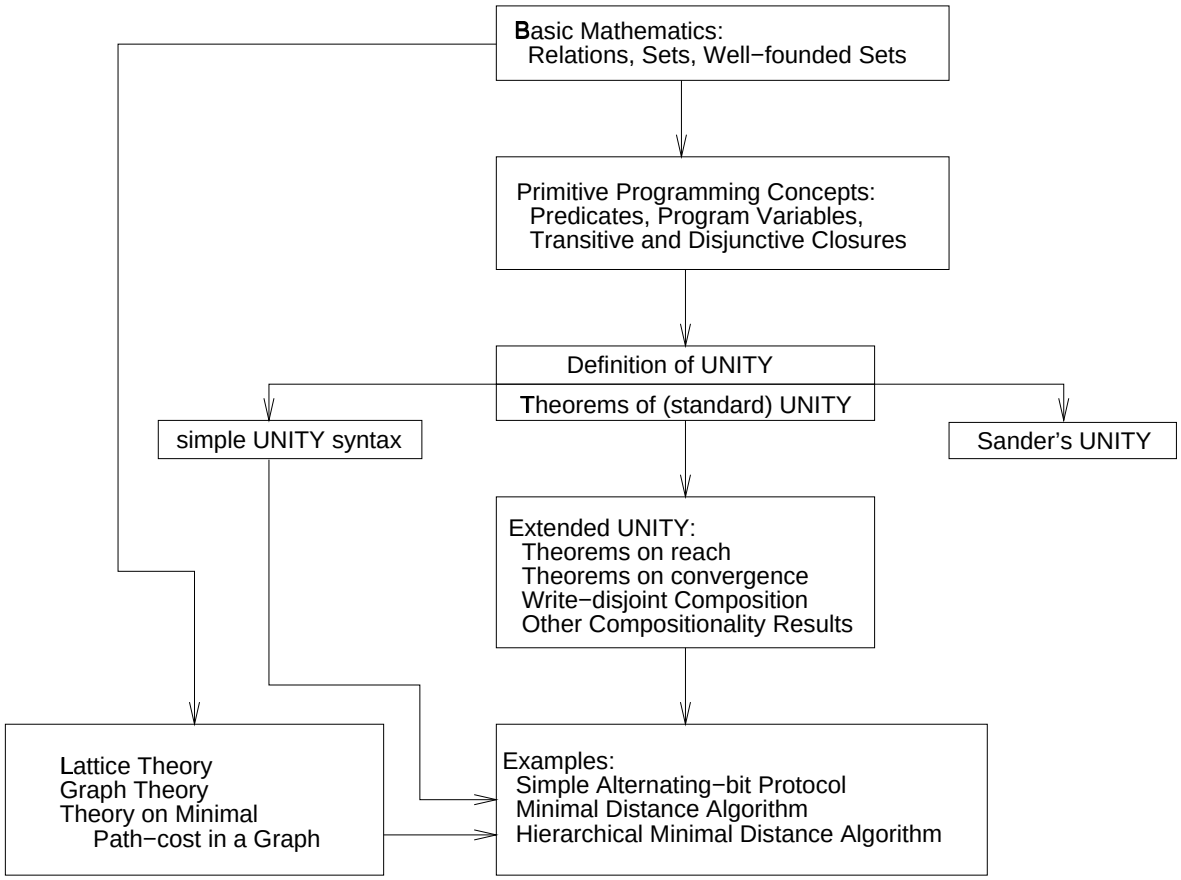

Figure 8: The structure of our mechanical verification work.

Work	*proof size*
Core UNITY	70 KB
Extension with $\rightarrowtail$, $\leadsto$, and Sander's extension	140 KB
Other theory required by application domain	540 KB
General theory of MinDist-like stabilization	120 KB
Hierarchical MinDist-like algorithms	110 KB

This paper as well as above mentioned results are part of an on going research at Utrecht University. Documented results include: Lentfert's thesis on hierarchical algorithms, Prasetya's thesis [20] on formal design and mechanical verification of distributed programs, and a recent work by Vos on extending UNITY embedding multi-typed values, refinement theory, and mechanical verification of various distributed algorithms [24]. The papers are all available at request. Prasetya's thesis is also available at:

`http://www.cs.uu.nl/docs/research/publication/Theses.html`

Part II
Examples

In Part I a theory of self-stabilizing systems has been presented. In this Part II we will demonstrate how the theory is applied. The derivation of three self-stabilizing algorithms will be shown. They are, in increasing complexity: leader election in a ring shaped network, minimum computation in a tree shaped network, and minimum distance computation in an arbitrarily connected network. In addition, in the third example inter-process communication will be made explicit. In general, addition of communication medium does not always preserve stabilization properties and therefore complicates the proof. So we want to show in what way the proof is influenced and in how far the proof can be separated between the core proof of the algorithm and the proof regarding the stabilization of the communication medium.

Major steps in the proof of the second and third examples are due to Lentfert and Swierstra [15, 14]. We worked them out further to make the calculation shorter and more natural.

For each example an implementation will be given but its proof will only be shown up to a certain point (beyond which it becomes mostly mechanical and contributes little to our understanding). No new algorithm is involved, which is also not the concern of this paper. All examples satisfy specifications of the form:

$$\mathsf{true} \ {}_P\vdash \mathsf{true} \leadsto q$$

where q describes the goal to which the program P must stabilize. The two true's above mean that P is self-stabilizing regardless *any* transient sabotages by an adversary. However, the specifications implicitly assume certain topologies of distributed systems. In the first example it is a ring; in the second a tree; and in the third an arbitrary connected network. It means that the stabilization is

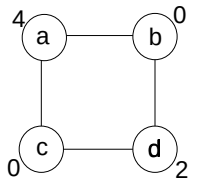

Figure 9: A simple network.

guaranteed even if the adversary changes the topology, provided the resulting topology still satisfies the assumed form.

A Note on Induction

Induction plays an especially important role in proving self-stabilization, but in many cases its application can be tricky. Consider for example the following program to compute minimal distance between nodes in a network:

prog MinDist
init true
assign $([]a : a \in V : d.a.a := 0)$
$[]$ $([]a, b : a, b \in V \wedge (a \neq b) : d.a.b := \mathsf{min}\{b' : b' \in E.b : d.a.b' + 1\})$

The initial condition true indicates that the program is self-stabilizing. Most of us will be ready to believe that this program does what it must do. It is a lucky guess though, for its formal proof is far from trivial. Misled by intuition one may conclude that, for example, the difference between each $d.a.b$ and the actual minimal distance between a and b gradually decreases and hence eventually $d.a.b$ will take the value of the actual minimal distance (notice that the argument is actually an instance of inductive decomposition). It sounds plausible, but it does not work. Consider the simple network in Figure 9.

The number printed outside a node i, $i \in \{a, b, c, d\}$, denotes the initial value of $d.a.i$. The difference between $d.a.a$ and the actual distance between a and a will indeed decrease, but the same things cannot be said for $d.a.d$, for example. It already contains the correct value, and if the assignment to $d.a.d$ is executed first, $d.a.d$ will 'deviate further' from the correct value rather than 'approaching' it.

Many proofs of self-stabilizing systems [1, 8, 14] exhibit this kind of subtle inductive decomposition. Handling induction informally will in this case greatly raise the risk of making mistakes and impede the discovery of mathematical properties of the problem which otherwise may lead to a better solution. So, what we also want to show in the examples given later is how to apply inductive decomposition of self-stabilization in a formal way, and straightforwardly.

Sections overview: Sections 9, 10, and 11 present the examples, one for each

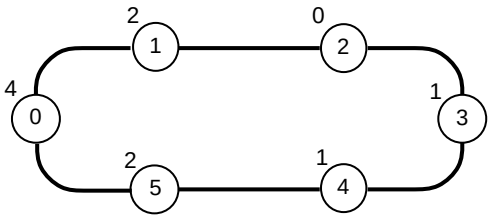

Figure 10: A ring network.

section.

A note on notation: for the sake of readability confinement requirements (that is, expressions of the form $p \in \mathsf{Pred}.V$ –see Part I for the meaning of it) will be omitted from formulas.

9 Leader Election

We have N processes numbered from 0 to $N-1$ connected in a ring: process i is connected to process i^+ where $^+$ is defined as $i^+ \;=\; (i+1) \bmod N$. *Node 0 is considered special.* Figure 10 shows such a ring of six processes.

Each process i has a local variable $x.i$ that contains a natural number less than N. For example, the numbers printed above the circles in Figure 10 show the values of the $x.i$'s of the corresponding processes. The problem is to make all processes agree on a common value of the $x.i$'s. The selected number is then the number of the 'leader' process, which is why the problem is called 'leader election'. The computation has to be self-stabilizing and non-deterministic. The latter means, for example as in the case shown in Figure 10, that the computation should not always choose 4 (the initial value of $x.0$) as the leader, or 0 (the minimum value of the $x.i$'s).

In this problem the adversary has the power to change the values of the $x.i$'s. For simplicity, the adversary is forbidden to change the topology of the network.

To do this we extend the $x.i$'s to range over natural numbers and allow them to have arbitrary initial values. The problem is generalized to computing a common value of $x.i$'s. The identity of the leader can be obtained by applying $\bmod N$ to the resulting common natural number.

Let us define a predicate ok as follows.

$$\mathsf{ok} \;=\; (\forall i : i < N : x.i = x.i^+)$$

The specification of the problem can be expressed as follows:

$$\mathsf{LS0}: \quad \mathsf{true} \;{}_{\mathsf{ring}}\!\vdash \mathsf{true} \leadsto \mathsf{ok}$$

Here is our strategy to solve the above. We let the value of $x.0$ decrease to a value which can no longer be *'affected'* by the value of other $x.i$'s —we choose to rule that only those $x.i$'s whose value is lower than $x.0$ *may* affect $x.0$. This value of $x.0$ is then propagated along the ring to be copied to each $x.i$ and hence we now have a common value of the $x.i$'s. Formally this is just an instance of the Bounded Progress principle with $x.0$ as the bound function.

Recall that the Bounded Progress principle states that if the program keep decreasing the value of some bounded function along some well-founded ordering, then it cannot do so forever. Let us now apply the principle to reflect the strategy. We calculate for LS0:

$$
\begin{array}{ll}
& \mathsf{true} \leadsto \mathsf{ok} \\
\Leftarrow & \quad \{ \text{ Definition of } \leadsto \ \} \\
& (\mathsf{true} \rightarrowtail \mathsf{ok}) \ \wedge \ (\ \circlearrowleft \mathsf{ok}) \\
\Leftarrow & \quad \{ \text{ Bounded Progress } \} \\
& (\forall M :: (x.0 = M) \rightarrowtail (x.0 < M) \vee \mathsf{ok}) \ \wedge \ (\ \circlearrowleft \mathsf{ok})
\end{array}
$$

For the progress part of above specification we derive further:

$$
\begin{array}{ll}
& (x.0 = M) \rightarrowtail (x.0 < M) \vee \mathsf{ok} \\
\Leftarrow & \quad \{ \rightarrowtail \text{ Disjunction } \} \\
& ((x.0 = M) \wedge \mathsf{ok} \rightarrowtail (x.0 < M) \vee \mathsf{ok}) \ \wedge \ ((x.0 = M) \wedge \neg \mathsf{ok} \rightarrowtail (x.0 < M) \vee \mathsf{ok}) \\
= & \quad \{ (\Rightarrow, \rightarrowtail) \text{ Introduction } \} \\
& (x.0 = M) \wedge \neg \mathsf{ok} \rightarrowtail (x.0 < M) \vee \mathsf{ok} \\
\Leftarrow & \quad \{ \rightarrowtail \text{ Substitution } \} \\
& (x.0 = M) \wedge \neg \mathsf{ok} \rightarrowtail (x.0 < M)
\end{array}
$$

The last specification above states that the value of $x.0$ *must* decrease while ok is not established. But if ok is not yet established then there must be some i such that $x.i \neq x.0$. A naive solution is to send the minimum value of the $x.i$'s to $x.0$ but this results a deterministic program which always chooses the minimum value of the $x.i$'s as the common value. So, we will have to try something else. We let each process copy its $x.i$ to $x.i^+$. In this way the value of some $x.i$ which is smaller —not necessarily the smallest possible— than $x.0$, if one exists, will eventually reach process 0. Of course it is possible that values larger than $x.0$ reach process 0 first, but process 0 simply will ignore these values.

Let ts be defined as follows:

$$\mathsf{ts} \ = \ N \ - \ \mathsf{max}\{n : (n \leq N) \wedge (\forall i : i < n : x.i = x.0) : n\} \tag{9.1}$$

Roughly, ts is the length of the tail segment of the ring whose elements are yet to be made equal to $x.0$. Note that according to the just described strategy the value of $x.0$ either remains the same or decreases. If it does not decrease, it will be copied to $x.1$, then to $x.2$, and so on. In doing so ts will be decreased. Note that $\mathsf{ts} = 0$ implies ok. This is, again, an instance of the Bounded Progress principle with ts as the bound function. The above strategy can be translated to the formal level. Continuing our calculation:

$$
\begin{array}{ll}
 & (x.0 = M) \wedge \neg \mathsf{ok} \mapsto (x.0 < M) \\
\Leftarrow & \quad \{ \text{BOUNDED PROGRESS} \} \\
 & (\forall K : K < N : \quad (x.0 = M) \wedge \neg \mathsf{ok} \wedge (\mathsf{ts} = K) \\
 & \qquad\qquad\qquad \mapsto \\
 & \qquad\qquad\qquad ((x.0 = M) \wedge \neg \mathsf{ok} \wedge (\mathsf{ts} < K)) \vee (x.0 < M)\ ) \\
\Leftarrow & \quad \{ \text{ENSURES to } \mapsto \text{ LIFTING} \} \\
 & (\forall K : K < N : \quad (x.0 = M) \wedge \neg \mathsf{ok} \wedge (\mathsf{ts} = K) \\
 & \qquad\qquad\qquad \text{ensures} \\
 & \qquad\qquad\qquad ((x.0 = M) \wedge \neg \mathsf{ok} \wedge (\mathsf{ts} < K)) \vee (x.0 < M)\ )
\end{array}
$$

So, to summarize, we come to the following refinement of LS0:

For all $M \in \mathbb{N}$ and $K < N$:

$$
\begin{array}{ll}
\mathsf{LS1.a}: & \circlearrowleft \mathsf{ok} \\
\mathsf{LS1.b}: & (x.0 = M) \wedge \neg \mathsf{ok} \wedge (\mathsf{ts} = K) \\
 & \text{ensures} \\
 & ((x.0 = M) \wedge \neg \mathsf{ok} \wedge (\mathsf{ts} < K)) \vee (x.0 < M)
\end{array}
$$

LS1.a states that once the processes agree on a common value, they maintain this situation. LS1.b states that if a common value has not been found, then either the length of the tail segment should become smaller, which can be achieved by copying the value of $x.i$ to $x.i^+$, or $x.0$ should decrease.

Without proof we give a program that satisfies the above specification.

prog ring
read $\{i : i < N : x.i\}$
write $\{i : i < N : x.i\}$
init true
assign **if** $x.(N-1) < x.0$ **then** $x.0 := x.(N-1)$
$[\!]$ $([\!]i : i < N-1 : x.(i+1) := x.i)$

Note that the common natural number will be computed non-deterministically: there is no way of predicting which of the initial values of the $x.i$'s will be picked as the common number.

10 Self-stabilizing Computation of Minimum

In this section an example of another sort of Bounded Progress, namely a principle called *Round Decomposition*, will be used. Recall that the principle states that a specification of the form:

$$\mathsf{true} \leadsto (\forall n : n \in A : q.n)$$

can be realized by dividing executions in rounds and requiring the system to converge to $q.n$ at each round n. The order in which the rounds are traversed

does not have to be linear, but it has to be acyclic, or in other words: the ordering is well-founded.

We have a finite, non-empty set of vertices V connected to form a tree with root α. The connectivity between the vertices is represented by a function S such that for any $i \in V$, $S.i$ is the set of all sons of i. If we define $S^0.a = \{a\}$ and $S^{n+1} = S \circ S^n$, we can define the 'transitive' closure of S (denoted by S^+) as $S^+.a = \cup\{i : 0 < i : S^i.a\}$ and the 'transitive and reflexive' closure of S (denoted by S^*) as $S^*.a = S^+.a \cup S^0.a$. The function S^* and S^+ describes the set of, respectively, descendants and proper descendants of a given vertex. We can regard S^+ as a relation by defining $i\ S^+\ j = i \in S^+.j$. This relation S^+ is well-founded, and obviously: $S^*.\alpha = V$.

Each process i has an input $x.i$ and the problem is to compute the minimum of the $x.i$'s of all vertices in V, or in other words, of all descendants of α. Let us however consider a more general problem. Instead of the standard minimum operator we consider the least upper bound operator $\sqcap$ (also called the 'cap' operator) of some given complete semi-lattice [5]. The problem can be stated as: compute $\sqcap\{i : i \in S^*.\alpha : x.i\}$. If we let the result to be stored in $y.\alpha$, the problem can be specified as follows:

$$\mathsf{M1}: \quad \mathsf{true}\ {}_P\vdash\ \mathsf{true} \leadsto (y.\alpha = \sqcap\{i : i \in S^*.\alpha : x.i\})$$

In this context the adversary is assumed to have the power to change the values of the input data in $x.i$'s. The adversary can also add or delete a process, but only if the resulting topology remains a tree with root α then a correct result can be guaranteed.

Let us first introduce some abbreviations which we will use later:

For all $i \in V$:

$$\begin{array}{lcl} \mathsf{ok}^i & = & (y.i = \sqcap\{j : j \in S^*.i : x.j\}) \\ \mathsf{preOk}^i & = & (\forall j : j \in S^+.i : \mathsf{ok}^j) \end{array}$$

ok^i states that process i has a *'correct'* value of $y.i$ and preOk^i states that all processes that 'precede' i, which here means being proper descendants of i, have correct values of their y's. Using ok the specification M1 can be nicely written as $\mathsf{true} \leadsto \mathsf{ok}^\alpha$. We will however consider a stronger specification M2 so as to make M1 fits in the round decomposition 'design scheme':

$$\mathsf{M2}: \quad \mathsf{true}\ {}_P\vdash\!\leadsto (\forall i : i \in V : \mathsf{ok}^i)$$

To establish ok our strategy is as follows. Suppose that somehow we can establish ok^j for all proper descendant j of i, then we might try to establish

[5] An equivalent approach would be to use an idempotent, commutative, and associative operator $\oplus$ instead of a semi-lattice.

ok^i using this knowledge. This is done repeatedly until ok^α is established. This sounds very much like round decomposition: V is the set of rounds, ordered by S^+, and ok^i is the goal of round i. The following calculation will make this apparent:

$$
\begin{array}{ll}
 & \mathsf{true} \ {}_P\vdash \mathsf{true} \leadsto (\forall i : i \in V : \mathsf{ok}^i) \\
\Leftarrow & \quad \{\ S^+ \text{ is well founded; } \textsc{Round Decomposition}\ \} \\
 & (\forall i : i \in V : (\forall j : j \in S^+.i : \mathsf{ok}^j) \ {}_P\vdash \mathsf{true} \leadsto \mathsf{ok}^i) \\
= & \quad \{\ \text{definition of } \mathsf{preOk}\ \} \\
 & (\forall i : i \in V : \mathsf{preOk}^i \ {}_P\vdash \mathsf{true} \leadsto \mathsf{ok}^i)
\end{array}
$$

Notice that the final specification reflects our strategy. Furthermore, we observe that the task of establishing ok^i can be delegated to process i, which we will call $P.i$. If we insist that for each $i \in V$, $\mathbf{w}P.i = \{y.i\}$ then $P = (\llbracket i : i \in V : P.i)$ consists of programs that are pair-wise write-disjoint, which is nice because we can now apply the Transparency principle. We continue the calculation:

$$
\begin{array}{ll}
 & \mathsf{preOk}^i \ {}_P\vdash \mathsf{true} \leadsto \mathsf{ok}^i \\
\Leftarrow & \quad \{\ P = (\llbracket j : j \in V : P.j);\ \leadsto \textsc{Transparency};\ \circlearrowleft \textsc{Compositionality}\ \} \\
 & ({}_P\vdash \circlearrowleft \mathsf{preOk}^i) \ \wedge \ (\mathsf{preOk}^i \ {}_{P.i}\vdash \mathsf{true} \leadsto \mathsf{ok}^i) \\
\Leftarrow & \quad \{\ \textsc{ensures} \text{ to } \rightarrowtail \textsc{Lifting};\ \circlearrowleft \textsc{Compositionality}\ \} \\
 & ({}_P\vdash \circlearrowleft \mathsf{preOk}^i) \ \wedge \ ({}_{P.i}\vdash \circlearrowleft (\mathsf{preOk}^i \wedge \mathsf{ok}^i)) \ \wedge \ ({}_{P.i}\vdash \mathsf{preOk}^i \ \mathsf{ensures} \ \mathsf{ok}^i)
\end{array}
$$

To summarize, we have refined M2 to the following specification:

Let $P = (\llbracket j : j \in V : P.j)$ such that $(\forall i : i \in V : \mathbf{w}(P.i) = \{y.i\})$. For all $i \in V$:

M3.a : ${}_P\vdash \circlearrowleft \mathsf{preOk}^i$
M3.b : ${}_{P.i}\vdash \circlearrowleft (\mathsf{preOk}^i \wedge \mathsf{ok}^i)$
M3.c : ${}_{P.i}\vdash \mathsf{preOk}^i \ \mathsf{ensures} \ \mathsf{ok}^i$

In particular, M3.c states that we have to establish ok^i from preOk^i. This can be done by computing $\sqcap\{i' : i' \in S^*.i : x.i'\}$ from $\sqcap\{j' : j' \in S^*.j : x.j'\}$ of all sons j of i. To do that we exploit the following equality (the proof of which is left to the reader):

$$
\begin{array}{l}
\sqcap\{i' : i' \in S^*.i : x.i'\} \\
= \\
\underbrace{(\sqcap\{j' : j' \in S^*.j_0 : x.j'\}) \ \sqcap \ \cdots \ \sqcap \ (\sqcap\{j' : j' \in S^*.j_n : x.j'\})}_{\text{for all sons } j_k \text{ of } i} \ \sqcap \ x.i
\end{array}
$$

preOk implies however that for all sons j of i, $y.j = \sqcap\{j' : j' \in S^*.j : x.j'\}$, and hence above can be replaced by:

$$\sqcap\{i' : i' \in S^*.i : x.i'\} \ = \ (\sqcap\{j : j \in S.i : y.j\}) \ \sqcap \ x.i$$

and hence ok^i can be established by the assignment:

$$y.i \;:=\; (\sqcap\{j : j \in S.i : y.j\}) \;\sqcap\; x.i$$

Without further proof we give now a program that satisfies M3 (hence also M1):

$P = (\llbracket i : i \in V : P.i)$ where for all $i \in V$, $P.i$ is defined as follows:

prog	$P.i$
read	$\{j : j \in V : y.j\} \cup \{x.i, y.i\}$
write	$\{y.i\}$
init	true
assign	$y.i := (\sqcap\{j : j \in S.i : y.j\}) \sqcap x.i$

11 Computing Minimal Distances

Recall again the program MinDist displayed early in Part I of this paper. The program computes the minimal distance between any two vertices a and b in a network described by (V, E). V is the set of all vertices in the network and the connectivity between vertices is described by $E \in V \to \mathcal{P}(V)$ such that $E.i$ describes the set of all neighbors of i. The network is assumed to be *connected*. The program (in UNITY notation) is redisplayed below:

prog	MinDist
read	$\{a, b : a, b \in V : d.a.b\}$
write	$\{a, b : a, b \in V : d.a.b\}$
init	true
assign	$(\llbracket a : a \in V : d.a.a := 0)$
$\llbracket$	$(\llbracket a, b : a, b \in V \wedge (a \neq b) : d.a.b := \mathsf{min}\{b' : b' \in E.b : d.a.b' + 1\})$

The actual minimal distance between a and b is denoted by $\delta.a.b$. The function δ is also characterized by the following property, which also tells us how to compute $\delta.a.b$ from the $\delta.a.b'$ of the neighbors b' of b:

Theorem 11.1 : For all $a, b \in V$ with $a \neq b$:

$$\delta.a.a = 0 \;\wedge\; \delta.a.b = \mathsf{min}\{b' : b' \in E.b : \delta.a.b' + 1\}$$

The program MinDist is required to compute and maintain δ. This specification can be expressed as follows:

MD0 : $\mathsf{true}\ {}_{\mathsf{MinDist}}\!\vdash \mathsf{true} \leadsto (\forall a, b : a, b \in V : d.a.b = \delta.a.b)$

In this problem the adversary is assumed to have the power to change the values of the input data in $d.a.b$'s, and later also the content of 'communication channels' between processes. The adversary can also change the topology of the

network, but only if the resulting topology remains connected then a correct result can be guaranteed.

Observe that upon reaching its fixpoint, the program MinDist will have the value of d satisfying the equation in Theorem 11.1, and since the equation characterizes δ, then d is equal to δ. The problem is however, how do we know that this program will ever reach its fixpoint, especially since it can start in an arbitrary state? To prove this we will have to construct and prove a scenario we think the program obeys while converging to its fixpoint.

Another point is communication between components. In MinDist as well as in the programs from the previous two examples, it is assumed that each (parallel) component can directly access the variable of neighboring components. In practice this is not always the case. Channels are usually used when direct access is not possible. Let us now explicitly add channels to MinDist. For the purpose of understanding the algorithm we gain little from this extra feature, but we have another purpose here, namely to show just how far the addition of channels (in general: communication sub-systems) influences our calculation and our correctness concern[6].

To keep it simple, we model channels with link registers. The mechanism is simple, but captures the idea of asynchronous communication adequately. Sending a message into a channel is modelled by writing to a link register. The receiver is not obliged to immediately fetch the message and the sender is free to send another message at any time. Consequently, it is possible that the sender overwrites a message which has not been 'received'. This models loss of messages. The program MinDist will look like:

prog	MinDist
init	true
assign	$(\llbracket a : a \in V : d.a.a := 0)$
$\llbracket$	$(\llbracket a, b : a, b \in V \wedge (a \neq b) : d.a.b := \mathsf{min}\{b' : b' \in E.b : r.a.b.b' + 1\})$
$\llbracket$	$(\llbracket a, b, b' : a, b, b' \in V \wedge b \in E.b' : r.a.b'.b := d.a.b)$

in which $r.a.b.b'$ is a link register between the process in b that maintains $d.a.b$ and the neighboring process b' that maintains $d.a.b'$. The assignment $r.a.b'.b := V$ represents the action of b to send a value V to the channel (link register) $r.a.b'.b$ connecting it to process b'. We will return to this later.

11.1 Decomposing the Specification

First of all, we observe from Theorem 11.1 that $\delta.a$ can be computed without any information about $\delta.a'$ for distinct a'. We can delegate this task to a component program.

[6] MinDist does work correctly with channels, but in general inserting communication sub-systems between processes does not always preserve correctness. In pathologic cases, channels may initially contain 'bad' values which keep circulating in the system and hence preventing it from stabilizing.

Let $\mathsf{MinDist} = (\|a : a \in V : \mathsf{MinDist}.a)$ where the $\mathsf{MinDist}.a$'s are pair-wise write-disjoint. Using the Transparency law we can delegate the computation of $\delta.a$ to $\mathsf{MinDist}.a$:

$$
\begin{array}{ll}
 & \mathsf{true}\ {}_{\mathsf{MinDist}}\!\vdash \mathsf{true} \leadsto (\forall a, b : a, b \in V : d.a.b = \delta.a.b) \\
\Leftarrow & \quad \{ \leadsto \textsc{Conjunction} \} \\
 & (\forall a : a \in V : \mathsf{true}\ {}_{\mathsf{MinDist}}\!\vdash \mathsf{true} \leadsto (\forall b : b \in V : d.a.b = \delta.a.b)) \\
\Leftarrow & \quad \{ \mathsf{MinDist} = (\|a : a \in V : \mathsf{MinDist}.a); \leadsto \textsc{Transparency} \} \\
 & (\forall a : a \in V : \mathsf{true}\ {}_{\mathsf{MinDist}.a}\!\vdash \mathsf{true} \leadsto (\forall b : b \in V : d.a.b = \delta.a.b))
\end{array}
$$

So, MD0 can be refined by MD1:

For all $a \in V$:

$$\mathsf{MD1}: \ \mathsf{true}\ {}_{\mathsf{MinDist}.a}\!\vdash \mathsf{true} \leadsto (\forall b : b \in V : d.a.b = \delta.a.b)$$

We will now divide the execution of each $\mathsf{MinDist}.a$ into rounds (the rounds are abstract, that is, we pretend as if they exist, but they do not have to explicitly appear in the program text). Let us say that upon finishing a round n the program $\mathsf{MinDist}.a$ establishes and maintains $q.n$ where:

$$q.n \ = \ (\forall b : b \in V : \delta.a.b \leq n \Rightarrow (d.a.b = \delta.a.b)) \tag{11.1}$$

Notice that upon reaching a sufficiently 'large' round n the program $\mathsf{MinDist}.a$ will have achieved its goal as specified in MD1. Since the program maintains each $q.m$, upon entering round n, $(\forall m : m < n : q.m)$ holds. The obligation of round n can be expressed by:

$$(\forall m : m < n : q.m) \leadsto q.n$$

This sounds reasonable. Unfortunately $(\forall m : m < n : q.m)$ does not provide enough information to establish $q.n$.

Suppose $\delta.a.b = n + 1$. To complete round $n + 1$ we must compute $\delta.a.b$ to be assigned to $d.a.b$. Theorem 11.1 suggests that we can compute this from $\delta.a.b'$ of *all* neighbors b' of b. Since we have passed round n before coming to round $n + 1$ we know that $\delta.a.b'$ of any neighbor b' such that $\delta.a.b' = n$ has been computed correctly and stored in $d.a.b'$. Unfortunately, nothing is known about the value of the $d.a.b'$ of other neighbors. Another problem is that the link registers may initially contain garbage values which may circulate through the system and prevent it from stabilizing. Obviously, there are more things which have to be done before a round can be declared completed. Before we write down the details, let us first see how far we can go without a complete knowledge of the obligations of each round.

Let us assume the existence of a finite domain A of rounds, ordered by a well founded ordering $\prec$. Later, we will have to come up with a concrete A and $\prec$.

Before we continue, let us introduce some abbreviations. In the definition below, d and r are (arrays of) program variables. The role of d should be clear

by now. $r.a.b'.b$ is the link register between vertices b and b'. It is intended to be a copy of $d.a.b$ for vertex b'.

Definition 11.2 : Let A be a finite set of rounds ordered by a well-founded relation $\prec$. For all $n \in A$, $a, b \in V$, and $f \in V \to A$:

$$\begin{array}{lcl} \mathsf{ok}_b^n.X & = & \text{"}X \textit{ is an acceptable value for round } n \textit{ and vertex } b\text{"} \\ \mathsf{dataOk}^n & = & (\forall b : b \in V : \mathsf{ok}_b^n.(d.a.b)) \\ \mathsf{comOk}^n & = & (\forall b, b' : b \in V \wedge b' \in E.b : \mathsf{ok}_b^n.(r.a.b'.b)) \\ \mathsf{preOk}^n & = & (\forall m : m \prec n : \mathsf{dataOk}^m \wedge \mathsf{comOk}^m) \end{array}$$

So, $\mathsf{ok}_b^n.(d.a.b)$ means the value $d.a.b$ is acceptable for b at round n and $\mathsf{ok}_b^n.(r.a.b'.b)$ means that the link register $r.a.b'.b$ contains an acceptable value for b at round n. The meaning of *'acceptable'* is left open for now, but in any case it is sufficient if:

$$\mathsf{FinishCondition}: \quad (\forall n : n \in A : \mathsf{dataOk}^n) \Rightarrow (\forall b : b \in V : d.a.b = \delta.a.b)$$

dataOk^n means that *all* $d.a.b$'s are acceptable for round n and comOk^n means the value of all link registers are also acceptable for round n. Finally, preOk^n means that the value of all $d.a.b$'s and their copies are acceptable for all rounds previous to round n.

Assuming Finish Condition we can refine MD1 to:

$$\mathsf{true} \ {}_{\mathsf{MinDist.}\,a}\!\vdash \mathsf{true} \leadsto (\forall n : n \in A : \mathsf{dataOk}^n) \tag{11.2}$$

The Round Decomposition principle suggests that we can implement (11.2) by converging to dataOk^n at each new round n, given that dataOk^m holds for any previous round m. Unfortunately dataOk does not tell anything about the state of the link registers. Without this knowledge we cannot tell anything about the result of a local computation of a node since it is based on the values of the link registers. So, what we do is strengthen (11.2) by requiring that the values of the link registers should also be made acceptable at each new round. This gives us the following specification:

$$\mathsf{true} \ {}_{\mathsf{MinDist.}\,a}\!\vdash \mathsf{true} \leadsto (\forall n : n \in A : \mathsf{dataOk}^n \wedge \mathsf{comOk}^n) \tag{11.3}$$

Let us now try to apply the Round Decomposition principle to refine (11.3) further. We derive:

$$\begin{array}{ll} & \mathsf{true} \vdash \mathsf{true} \leadsto (\forall n : n \in A : \mathsf{dataOk}^n \wedge \mathsf{comOk}^n) \\ \Leftarrow & \quad \{ \text{ Definition of } \mathsf{preOk};\ \textsc{Round Decomposition} \ \} \\ & (\forall n : n \in A : \mathsf{preOk}^n \vdash \mathsf{true} \leadsto \mathsf{dataOk}^n \wedge \mathsf{comOk}^n) \\ \Leftarrow & \quad \{ \ \textsc{Accumulation} \ \} \\ & (\forall n : n \in A : \ (\mathsf{preOk}^n \vdash \mathsf{true} \leadsto \mathsf{dataOk}^n) \wedge \\ & \qquad\qquad\qquad\quad (\mathsf{preOk}^n \vdash \mathsf{dataOk}^n \leadsto \mathsf{comOk}^n) \ ) \end{array}$$

$\Leftarrow$ { $\leadsto$ Stable Shift }

$(\forall n : n \in A :$ $(\mathsf{preOk}^n \vdash \mathsf{true} \leadsto \mathsf{dataOk}^n) \wedge$
$(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \vdash \mathsf{true} \leadsto \mathsf{comOk}^n)\ )$

So, MD1 can be refined by MD2 and MD3 defined as follows:

For all $n \in A$:

MD2.a : $\mathsf{preOk}^n\ {}_{\mathsf{MinDist}.a}\vdash \mathsf{true} \leadsto \mathsf{dataOk}^n$

MD2.b : $\mathsf{preOk}^n \wedge \mathsf{dataOk}^n\ {}_{\mathsf{MinDist}.a}\vdash \mathsf{true} \leadsto \mathsf{comOk}^n$

By unfolding the definition of dataOk and comOk and by applying the $\leadsto$ Conjunction law we can refine above specifications to the following. For all $n \in A$, $b \in V$, and $b' \in E.b$:

$$\mathsf{preOk}^n\ {}_{\mathsf{MinDist}.a}\vdash \quad \mathsf{true} \leadsto \mathsf{ok}^n_b.(d.a.b) \tag{11.4}$$

$$\mathsf{preOk}^n \wedge \mathsf{dataOk}^n\ {}_{\mathsf{MinDist}.a}\vdash \quad \mathsf{true} \leadsto \mathsf{ok}^n_b.(r.a.b'.b) \tag{11.5}$$

Let us insist that $\mathsf{MinDist}.a = (\llbracket b : b \in V : \mathsf{MinDist}.a.b)$ where the $\mathsf{MinDist}.a.b$'s are pair-wise write-disjoint. Using the Transparency law we can delegate the task of fulfilling (11.4) and (11.5) to $\mathsf{MinDist}.a.b$. If we do this, we end up with the following refinement of MD2:

For all $n \in A, b \in V$, and $b' \in E.b$:

MD3.a : $\mathsf{preOk}^n\ {}_{\mathsf{MinDist}.a.b}\vdash \mathsf{true} \leadsto \mathsf{ok}^n_b.(d.a.b)$

MD3.b : $\mathsf{preOk}^n \wedge \mathsf{dataOk}^n\ {}_{\mathsf{MinDist}.a.b}\vdash \mathsf{true} \leadsto \mathsf{ok}^n_b.(r.a.b'.b)$

Let us first continue with MD3.b since this is easier. By applying ensures to $\leadsto$ Lifting we can refine MD3.b to the following primitive level specifications:

$${}_{\mathsf{MinDist}.a.b}\vdash \quad \circlearrowleft(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n) \tag{11.6}$$

$${}_{\mathsf{MinDist}.a.b}\vdash \quad \circlearrowleft(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \wedge \mathsf{ok}^n_b.(r.a.b'.b)) \tag{11.7}$$

$${}_{\mathsf{MinDist}.a.b}\vdash \quad (\mathsf{preOk}^n \wedge \mathsf{dataOk}^n)\ \mathsf{ensures}\ \mathsf{ok}^n_b.(r.a.b'.b) \tag{11.8}$$

The progress specification (11.8) can be simplified using (11.6):

$(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n)\ \mathsf{ensures}\ \mathsf{ok}^n_b.(r.a.b'.b)$

$\Leftarrow$ { second argument of ensures can be weakened [7] }

$(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n)\ \mathsf{ensures}\ (\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \wedge \mathsf{ok}^n_b.(r.a.b'.b)$

$\Leftarrow$ { definition of dataOk; weakening the second argument of ensures }

$(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n)\ \mathsf{ensures}\ (\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \wedge (r.a.b'.b = d.a.b))$

$\Leftarrow$ { ensures PSP law; (11.6) }

true ensures $(r.a.b'.b = d.a.b)$

Notice that the resulting ensures specification can be implemented by an assignment $r.a.b'.b := d.a.b$.

Now let us continue with MD3.a. Just as what we did to MD3.b, we apply ensures to $\rightsquigarrow$ LIFTING to refine MD3.a to:

$$_{\mathsf{MinDist}.a.b}\vdash \quad \circlearrowleft \mathsf{preOk}^n \tag{11.9}$$

$$_{\mathsf{MinDist}.a.b}\vdash \quad \circlearrowleft (\mathsf{preOk}^n \wedge \mathsf{ok}^n_b.(d.a.b)) \tag{11.10}$$

$$_{\mathsf{MinDist}.a.b}\vdash \quad \mathsf{preOk}^n \text{ ensures } \mathsf{ok}^n_b.(d.a.b) \tag{11.11}$$

The progress specification (11.11) requires the program to establish $\mathsf{ok}^n_b.(d.a.b)$ from preOk^n. The latter implies that the link registers $r.a.b'.b$ of all $b' \in E.b$ are all ok for any previous round m. We might be able to establish $\mathsf{ok}^n_b.(d.a.b)$ by applying some function θ to those link registers. Let us suppose that there exists such a function. More specifically, assume:

There exists a function θ satisfying:

$$\mathsf{RS}: \quad (\forall m, b' : m \prec n \ \wedge \ b' \in E.b : \mathsf{ok}^m_{b'}.(f.b')) \ \Rightarrow \ \mathsf{ok}^n_b.(\theta^b_a.f)$$

for all $n \in A$ and $a, b \in V$ and f.

Let us now see how we can use RS to simplify (11.11). The calculation is similar to that of (11.8):

preOk^n ensures $\mathsf{ok}^n_b.(d.a.b)$

$\Leftarrow$ { second argument of ensures can be weakened [7] }

preOk^n ensures $(\mathsf{preOk}^n \wedge \mathsf{ok}^n_b.(d.a.b))$

$\Leftarrow$ { definition of preOk; use RS and choose $f \leftarrow r.a.b$; weakening second argument of ensures }

preOk^n ensures $(\mathsf{preOk}^n \wedge (d.a.b = \theta^b_a(r.a.b)))$

$\Leftarrow$ { ensures PSP law; (11.9) }

true ensures $(d.a.b = \theta^b_a(r.a.b))$

The last can be implemented by an assignment $d.a.b := \theta^b_a(r.a.b)$.

So, to summarize, we can refine MD3.a and MD3.b to:

MD3.a.1	$_{\mathsf{MinDist}.a.b}\vdash \circlearrowleft \mathsf{preOk}^n$
MD3.a.2	$_{\mathsf{MinDist}.a.b}\vdash \circlearrowleft (\mathsf{preOk}^n \wedge \mathsf{ok}^n_b.(d.a.b))$
MD3.a.3	$_{\mathsf{MinDist}.a.b}\vdash \mathsf{true}$ ensures $(d.a.b = \theta^b_a.(r.a.b))$
MD3.b.1	$_{\mathsf{MinDist}.a.b}\vdash \circlearrowleft (\mathsf{preOk}^n \wedge \mathsf{dataOk}^n)$
MD3.b.2	$_{\mathsf{MinDist}.a.b}\vdash \circlearrowleft (\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \wedge \mathsf{ok}^n_b.(r.a.b'.b))$
MD3.b.3	$_{\mathsf{MinDist}.a.b}\vdash \mathsf{true}$ ensures $(r.a.b'.b = d.a.b)$

Without further proof we give a complete code for MinDist which satisfies the above specification:

MinDist $= (\llbracket a, b : a, b \in V : \mathsf{MinDist}.a.b)$ where $\mathsf{MinDist}.a.b$ is defined as follow:

prog	$\mathsf{MinDist}.a.b$
read	$\{b' : b' \in E.b : r.a.b.b'\} \cup \{b' : b \in E.b' : r.a.b'.b\} \cup \{d.a.b\}$
write	$\{b' : b \in E.b' : r.a.b'.b\} \cup \{d.a.b\}$
init	true
assign	$d.a.b := \theta^b_a.(r.a.b) \ \ [\!] \ \ (\llbracket b' : b \in E.b' : r.a.b'.b := d.a.b)$

11.1.1 Round Solvable Problems

The program above is actually a solution for a class of problems, instead of just the minimal distance problem. Note that in arguing about the correctness of the program we only relied on the existence of a set of A of rounds, ordered by a well-founded ordering $\prec$, and a function θ satisfying the property RS. So, as long as we can find these $(A, \prec)$ and θ for a given notion of acceptability (the predicate ok) the program above is a solution of the problem. Problems that can be solved this way are called *Round Solvable* [14].

As for the minimum distance problem, the following instantiation will satisfy Finish Condition and RS. The proof of this can be found in, for example [14]. We have also mechanically verified this result. For A we choose the set of natural numbers less than or equal to the diameter of the network (V, E). $\prec$ is instantiated to $<$, and ok and θ are defined as:

Definition 11.3 :
For all $a, b \in V$ and $n \in A$:

$$\mathsf{ok}^n_b.X \ = \ \begin{cases} X = \delta.a.b & \text{if } \delta.a.b \leq n \\ n \leq X & \text{otherwise} \end{cases}$$

Definition 11.4 :
For any $f \in V \rightarrow \mathbb{N}$ and $a, b \in V$:

$$\theta^b_a.f \ = \ \begin{cases} 0 & \text{if } a = b \\ (\mathsf{min} b' : b' \in E.b : f.b') + 1 & \text{otherwise} \end{cases}$$

References

[1] Y. Afek and G.M. Brown. Self-stabilization of the alternating-bit protocol. In *Proceeding of the IEEE 8th Symposium on Reliable Distributed Systems*, 1989.

[2] Flemming Andersen. *A Theorem Prover for UNITY in Higher Order Logic.* PhD thesis, Technical University of Denmark, 1992.

[3] A. Arora. *A foundation for fault-tolerant computing.* PhD thesis, Dept. of Comp. Science, Univ. of Texas at Austin, 1992.

[4] A. Arora and M.G. Gouda. Distributed reset. In *Proceedings of the 10th Conference on Foundation of Software Technology and Theoretical Computer Science*, 1990. Also in *Lecture Notes on Computer Science* vol. 472.

[5] A. Arora and M.G. Gouda. Closure and convergence: A foundation for fault-tolerant computing. In *Proceedings of the 22nd International Conference on Fault-Tolerant Computing Systems*, 1992.

[6] J.E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Trans. Programming Language Systems*, 11(2):330–344, 1989.

[7] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation.* Addison-Wesley Publishing Company, Inc., 1988.

[8] N.S. Chen, H.P. Yu, and S.T. Huang. A self-stabilizing algorithm for constructing spanning trees. *Information Processing Letters*, 39(3):147–151, 1991.

[9] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, 1974.

[10] Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[11] M.G. Gouda and T. Herman. Addaptive programming. *IEEE Trans. Software Eng.*, 17(9), September 1991.

[12] Ted Herman. *Adaptivity through Distributed Convergence.* PhD thesis, University of Texas at Austin, 1991.

[13] H.S.M. Kruijer. Self-stabilization (in spite of distributed control) in tree structured systems. *Information Processing Letters*, 2(8):91–95, 1979.

[14] P.J.A. Lentfert. *Distributed Hierarchical Algorithms.* PhD thesis, Utrecht University, April 1993.

[15] P.J.A. Lentfert and S.D. Swierstra. Towards the formal design of self-stabilizing distributed algorithms. In P. Enjalbert, A. Finkel, and K.W. Wagner, editors, *STACS 93, Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, pages 440–451. Springer-Verlag, February 1993.

[16] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems—Specification.* Springer-Verlag, 1992.

[17] I.S.W.B. Prasetya. Formalization of variables access constraints to support compositionality of liveness properties. In J.J. Joyce and C.J.H. Seger, editors, *LNCS 780: Higher Order Logic Theorem Proving and Its Applications*, pages 324–337. Springer-Verlag, 1993.

[18] I.S.W.B. Prasetya. *Lifted Predicate Calculus in HOL*. University of Utrecht, 1993. draft version.

[19] I.S.W.B Prasetya. *UU_UNITY: a Mechanical Proving Environment for UNITY Logic*. University of Utrecht, 1993. Draft. Available at request.

[20] I.S.W.B. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms*. PhD thesis, Dept. of Comp. Science, Utrecht University, 1995.

[21] B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[22] A.K. Singh. Leads-to and program union. *Notes on UNITY*, 06-89, 1989.

[23] R. Udink, T. Herman, and J. Kok. Compositional local progress in unity. In *proceeding of IFIP Working Conference on Programming Concepts, Methods and Calculi,*, 1994.

[24] T. Vos. *UNITY in Diversity: A Stratified Approach to the Verification of Distributed Algorithms*. PhD thesis, Informatica Instituut, Universiteit Utrecht, 2000.