[Her91] Ted Herman. *Adaptivity through Distributed Convergence.* PhD thesis, University of Texas at Austin, 1991.

[Kru79] H.S.M. Kruijer. Self-stabilization (in spite of distributed control) in tree structured systems. *Information Processing Letters*, 2(8):91–95, 1979.

[Len93] P.J.A. Lentfert. *Distributed Hierarchical Algorithms.* PhD thesis, Utrecht University, April 1993.

[LS93] 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.

[Pra93a] I.S.W.B. Prasetya. Formalization of variables access constraints to support compositionality of liveness properties. In *Proceeding HUG 93, HOL User's Group Workshop*, pages 326–339. University of British Columbia, 1993.

[Pra93b] I.S.W.B Prasetya. *UU_UNITY: a Mechanical Proving Environment for UNITY Logic.* University of Utrecht, 1993. Draft. Available at request.

[Pra94] I.S.W.B. Prasetya. Towards a mechanically supported and compositional calculus to design distributed algorithms. In T.F. Melham and J. Camilleri, editors, *Lecture Notes in Computer Science 859: Higher Order Theorem Proving and Its Application*, pages 362–377. Springer-Verlag, 1994.

[Pre93] I.S.W.B. Prestya. *Lifted Predicate Calculus in HOL.* University of Utrecht, 1993. draft version.

[San91] B.A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, 1991.

[Sin89] A.K. Singh. Leads-to and program union. *Notes on UNITY*, 06-89, 1989.

[UHK94] 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.

Figure 13 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 [San91], 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. It also requires a lot of knowledge of 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 and string are available in HOL. The most work in the verification of MinDist is *not* in the application of the programming logic —which we estimate only covers 10% of the total work— 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.

References

[AB89a] Y. Afek and G.M. Brown. Self-stabilization of the alternating-bit protocol. In *IEEE 8th Symposiom on Reliable Distributed Systems*, October 1989.

[AB89b] Y. Afek and G.M. Brown. Self-stabilization of the alternating-bit protocol. In *Proceeding of the IEEE 8th Symposium on Reliable Distributed Systems*, pages 80–83, 1989.

[AG90] 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.

[AG92] 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.

[And92] Flemming Andersen. *A Theorem Prover for UNITY in Higher Order Logic*. PhD thesis, Technical University of Denmark, 1992.

[BP89] J.E. Burns and J. Pachl. Uniform self-stabilizing rings. *ACM Trans. Programming Language Systems*, 11(2):330–344, 1989.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design – A Foundation*. Addison-Wesley Publishing Company, Inc., 1988.

[CYH91] 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.

[Dij74] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, 1974.

[DIM90] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computation*, pages 119–132, August 1990.

[GH91] M.G. Gouda and T. Herman. Addaptive programming. *IEEE Trans. Software Eng.*, 17(9), September 1991.

[GM93] Mike J.C. Gordon and Tom F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

the standard definition of UNITY and hence cannot provide the compositionality results that we have. The package does not provide laws for self-stabilization either.

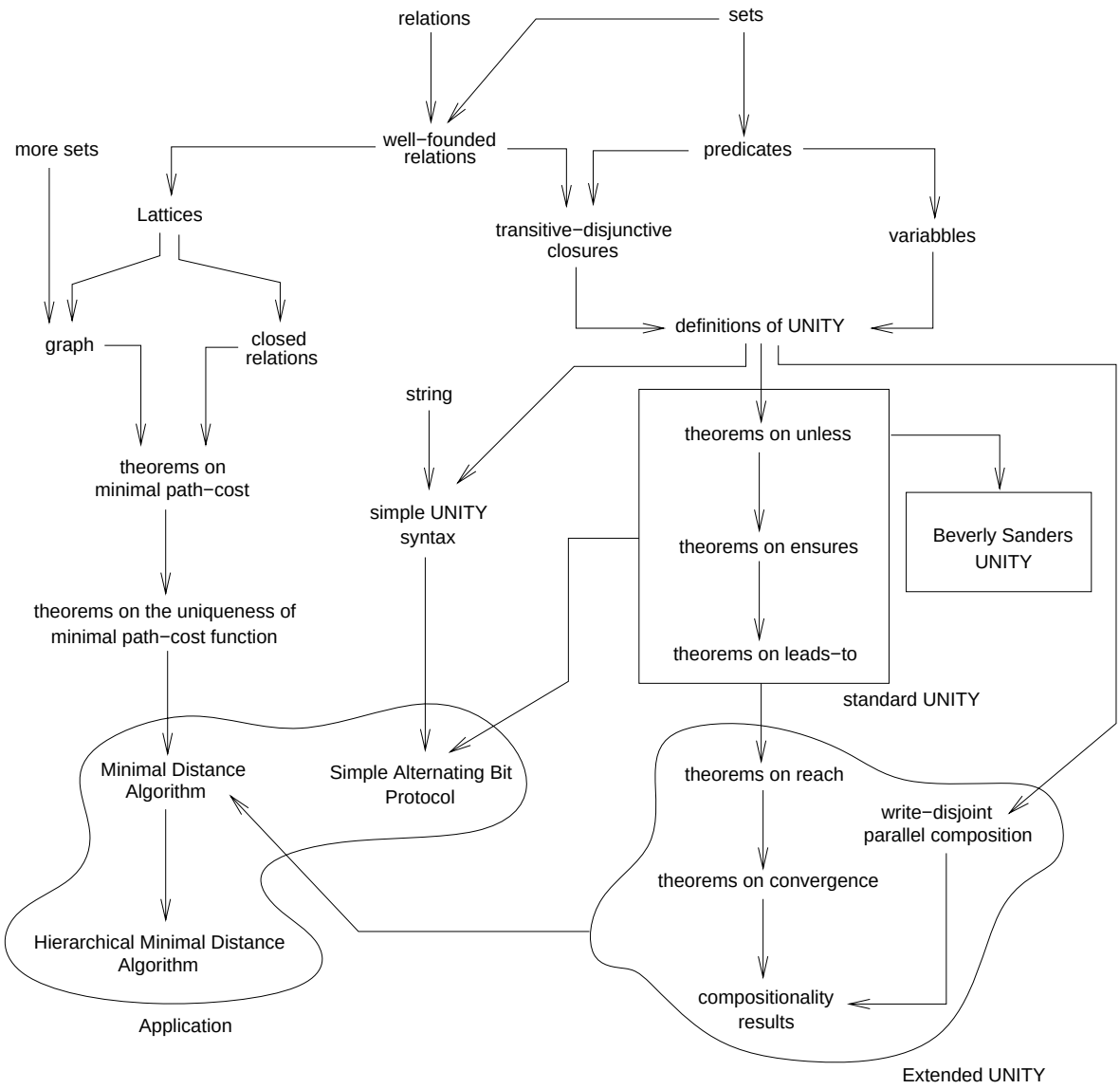

Figure 13: The structure of our mechanical verification work.

The transformation does not preserve (8.4). If initially $a \neq b$, consider the execution:

$(a_1; a_3; a_2; a_4)^*$

The execution is fair, but $a = b$ will never happen. The transformation does not preserve (8.5) either since the described stability can be destroyed by a_1 or a_2. This problem remains even if we use other models for the channels (such as queues).

The example above has a broader implication. For example suppose we want to replace the link registers in the program MinDist from Subsection 7.5 with queues. A queue can be viewed as a sequence (with variable length) of link registers. Consequently we will have the same problem as in the example above. For example, specification MD4.1 and MD4.2 will no longer be valid. Still, let us take a closer look. We have refined the original specification MD0 into the pair:

$$\begin{array}{ll} \mathsf{MD2.a}: & \mathsf{preOk}^n \ {}_{P.a}\vdash \mathsf{true} \leadsto \mathsf{dataOk}^n \\ \mathsf{MD2.b}: & \mathsf{preOk}^n \wedge \mathsf{dataOk}^n \ {}_{P.a}\vdash \mathsf{true} \leadsto \mathsf{comOk}^n \end{array}$$

MD2.a states that the system must establish and maintain acceptable values for the current round, assuming the system, including all communication means, has been stabilized for all previous rounds. MD2.b states that the system must also stabilize its communication means. So far the calculation is independent of the exact definition of comOk, which is the only part of the specification that depends on the selected method of communication. We can at this point choose for the queues model and instantiate the definition for comOk accordingly. Then we proceed with the calculation. Unfortunately this means that we have to re-do some part of the original calculation (in the link registers model). It would be indeed easier, as we have remarked it before, if we can just transform back and forth between various models of communication. So far, however, there are not much results in this direction that we are aware of. People still handle the problem of self-stabilization in an asynchronous distributed system as a separate problem, for example as in the work of Dolev, Israeli, and Moran [DIM90].

9 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 that we 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 attemp, such as in [LS93], 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. We have presented three examples of increasing complexity, with which we demonstrated the application of various laws introduced in this paper, and with which we also showed a few examples of the afore mentioned translation. We hope the reader has learned something from them.

All displayed theorems and corollaries have also been mechanically verified with theorem prover HOL. HOL is a general purpose theorem prover based on Higher Order Logic. The logic is extensible and hence all our mechanized theorems can be re-used to mechanically verify other programs. For example, they have been used to mechanically verify the program MinDist from Subsection subsec.dist. Our complete HOL library is available at request[8].

Currently, we are working on a thesis on mechanical verification of distributed programs. The thesis should be available soon. We are also working on the verification of a hierarchical version of the program MinDist. A hierarchical network is considered instead of an ordinary one. In such a network, vertices are grouped into domains, and the domains are ordered in a tree-hierarchy. The program is part of a routing program for such a network of computers. There is a notion of visibility, defining which domains are visible from another. The intention is to reduce the amount of routing information which needs to be locally present in each node by requiring that the node needs only to be aware of visible domains.

[8] There is a standard UNITY library for HOL written by F. Anderssen [And92], but the package only supports

restriction which may be imposed by the program. For example, in the case of program MinDist the the graph of the processes is required to remain connected.

8 A Note on the Implementation

In some parallel system, a process can directly access the state of another process. Such a system has thus a memory space which is basically accessible for all processes (so called *shared memory system*). In fact this is the memory model of UNITY. Designing programs for such a system is thus rather straightforward in UNITY. Comparing the examples in Subsection 7.4 and 7.5 will illustrate this. However, many distributed systems do not have this property as a process can only access the state of another process by means of a channel. Such a system is however write-disjoint. This means that we can apply the compositionality results in Sections 6 to decompose a given specification into a set of local specifications of each (write-disjoint) process. One often implements such a process sequentially. A typical example is:

program i
upon the receiving of data x from j do:
    add j to P.i ;
    send an ackowledgement to j ;
    broadcast x to all neighbors not in P.i ;
    ...

We can transform a progress specification for such a program into a Hoare triple specification and contionue the calculation using the much simpler sequential programming logic. A typical specification of a component program has the form:

$$J \ {}_P\vdash \mathsf{true} \rightarrowtail q \tag{8.1}$$

which states that given that the input variables of P satisfies J, P will eventually establish q. If P has the form $(S_0; S_2; \ldots; S_{n-1})^*$, then (8.1) is met if the S's satisfy:

$$(\forall i : i < n : \{J\}\ S_i\ \{J\}) \tag{8.2}$$

$$\{J\}\ (S_0; S_2; \ldots; S_{n-1})\ \{q\} \tag{8.3}$$

where $\{p\}\ S\ \{q\}$ means total correctness.

That we can use sequential programming logic to deal with the sequential part of the program is fine. Still, the addition of channels makes reasoning about the program, in some respect, more complicated. It may be easier to first design an (higher level) program that assumes shared memory, then transform it to a program that communicates with channels. A simple example below shows that such a transformation does not always preserve a given specification.

Consider a composite program $P[\!]Q$ where P and Q each consists of the following single actions:

$$P : a := b \quad \text{and} \quad Q : b := a$$

The program $P[\!]Q$ satisfies:

$$\mathsf{true} \ {}_{P[\!]Q}\vdash \quad \mathsf{true} \rightarrowtail (a = b) \tag{8.4}$$

$$ {}_{P[\!]Q}\vdash \quad \circlearrowleft (a = b) \tag{8.5}$$

Let us now add (asynchronous) channels to $P[\!]Q$. Like in the example in Subsection 7.5, we use link registers to model channels. Link register $r.a$ will be the Q's copy of a and $r.b$ will be the P's copy of b. So now, for example, P can only read b indirectly via $r.b$. This gives us the following program:

$$\begin{array}{ll} (a_1) & a := r.b \quad [\!] \\ (a_2) & r.a := a \quad [\!] \\ (a_3) & b := r.a \quad [\!] \\ (a_4) & r.b := b \end{array}$$

$\mathsf{MinDist} = (\llbracket a, b : a, b \in V : \mathsf{MinDist}.a.b)$ 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	$d.a.b := \theta^b_a.(r.a.b) \ \llbracket \ (\llbracket b' : b \in E.b' : r.a.b'.b := d.a.b)$

Round Solvable Problems

The program above is self-stabilizing: if an adversary corrupts the values of d or r the program simply continues its computation and eventually the value of every $d.a.b$ will become acceptable for all rounds in A. When this situation is reached, our notion of acceptability must be strong enough to conclude that the original goal of the computation is reached —in this case computing the minimal distance between vertices. This requirement is stated in Finish Condition.

The program above is actually apt 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* [Len93].

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 [Len93]. 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 7.8 :
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 7.9 :
For any $f \in V \rightarrow \mathbb{N}$ and $a, b \in V$:

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

7.6 Dynamic System

Let us here add a little comment about dynamic system. In a distributed and *dynamic system*, it is possible that a network of processes changes during the lifetime of the system: a process may temporarily cease to function or a new process may be created. The same thing may also happen to a link between two processes.

A self-stabilizing program can be parameterized by the topology of the network, as is the case with the program MinDist. If we can show that the program converges to its goal under any topology then the program can recover from a topological change. If after some time there is a change in the topology the computation will be temporarily in chaos while the environment is busy updating various topology-related data in, deleting suspended processes from, and adding new or re-activated processes to the program. Once this is done, the program simply continues its computation. Since the program has been shown to converge to its goal given, roughly speaking, any topology, it will also do so now. Of course, we assume that the new topology satisfies the

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

$\Leftarrow$ { ensures Post-weakening (analogous to Theorem 4.13 }

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

$\Leftarrow$ { definition of dataOk; ensures Post-weakening }

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

$\Leftarrow$ { ensures PSP law; (7.13) }

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

Notice that the resulting ensures specification can be implemented by an assignment $r.a.b'.b := d.a.b$. To summarize, we can refine MD3.b to:

MD4.a	$_{\mathsf{MinDist}.a.b}\vdash \circlearrowleft(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n)$
MD4.b	$_{\mathsf{MinDist}.a.b}\vdash \circlearrowleft(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \wedge \mathsf{ok}^n_b.(r.a.b'.b))$
MD4.c	$_{\mathsf{MinDist}.a.b}\vdash$ true ensures $(r.a.b'.b = d.a.b)$

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

$$_{\mathsf{MinDist}.a.b}\vdash \quad \circlearrowleft \mathsf{preOk}^n \tag{7.16}$$

$$_{\mathsf{MinDist}.a.b}\vdash \quad \circlearrowleft(\mathsf{preOk}^n \wedge \mathsf{ok}^n_b.(d.a.b)) \tag{7.17}$$

$$_{\mathsf{MinDist}.a.b}\vdash \quad \mathsf{preOk}^n \text{ ensures } \mathsf{ok}^n_b.(d.a.b) \tag{7.18}$$

The progress specification (7.18) 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:

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

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

Let us now see how we can use RS to simplify (7.18). The calculation is similar to that of (7.15):

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

$\Leftarrow$ { ensures Post-weakening }

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

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

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

$\Leftarrow$ { ensures PSP law; (7.16) }

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

The last can be implemented by an assignment $d.a.b := \theta^b_a(r.a.b)$. To summarize, we can refine MD3.a to:

MD5.a	$_{\mathsf{MinDist}.a.b}\vdash \circlearrowleft \mathsf{preOk}^n$
MD5.b	$_{\mathsf{MinDist}.a.b}\vdash \circlearrowleft(\mathsf{preOk}^n \wedge \mathsf{ok}^n_b.(d.a.b))$
MD5.c	$_{\mathsf{MinDist}.a.b}\vdash$ true ensures $(d.a.b = \theta^b_a.(r.a.b))$

Without further proof we give a code for MinDist which satisfies MD4 and MD5:

Assuming Finish Condition we can refine MD1 to:

$$\mathsf{true}\ {}_{\mathsf{MinDist}.a}\!\vdash\ \mathsf{true} \leadsto (\forall n : n \in A : \mathsf{dataOk}^n) \tag{7.9}$$

The Round Decomposition principle suggests that we can implement (7.9) 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 (7.9) by requiring that the values of the link registers should also be made acceptable at each new round. This gives us the following specification:

$$\mathsf{true}\ {}_{\mathsf{MinDist}.a}\!\vdash\ \mathsf{true} \leadsto (\forall n : n \in A : \mathsf{dataOk}^n \wedge \mathsf{comOk}^n) \tag{7.10}$$

Let us now try to apply the Round Decomposition principle to refine (7.10) further. We derive:

$\mathsf{true} \vdash \mathsf{true} \leadsto (\forall n : n \in A : \mathsf{dataOk}^n \wedge \mathsf{comOk}^n)$
$\Leftarrow$ { Definition of preOk; Round Decomposition }
$(\forall n : n \in A : \mathsf{preOk}^n \vdash \mathsf{true} \leadsto \mathsf{dataOk}^n \wedge \mathsf{comOk}^n)$
$\Leftarrow$ { Accumulation }
$(\forall n : n \in A : (\mathsf{preOk}^n \vdash \mathsf{true} \leadsto \mathsf{dataOk}^n) \ \wedge\ (\mathsf{preOk}^n \vdash \mathsf{dataOk}^n \leadsto \mathsf{comOk}^n))$
$\Leftarrow$ { $\leadsto$ 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 (Theorem 5.11) we can refine above specifications to the following. For all $n \in A$, $b \in V$, and $b' \in E.b$:

$$\mathsf{preOk}^n\ {}_{\mathsf{MinDist}.a}\!\vdash\ \ \mathsf{true} \leadsto \mathsf{ok}_b^n.(d.a.b) \tag{7.11}$$
$$\mathsf{preOk}^n \wedge \mathsf{dataOk}^n\ {}_{\mathsf{MinDist}.a}\!\vdash\ \ \mathsf{true} \leadsto \mathsf{ok}_b^n.(r.a.b'.b) \tag{7.12}$$

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 (Theorem 6.7) we can delegate the task of fulfilling (7.11) and (7.12) 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 : $\mathsf{preOk}^n\ {}_{\mathsf{MinDist}.a.b}\!\vdash\ \mathsf{true} \leadsto \mathsf{ok}_b^n.(d.a.b)$
MD3.b : $\mathsf{preOk}^n \wedge \mathsf{dataOk}^n\ {}_{\mathsf{MinDist}.a.b}\!\vdash\ \mathsf{true} \leadsto \mathsf{ok}_b^n.(r.a.b'.b)$

Let us first continue with MD3.b since this is easier. By applying (ensures,$\leadsto$) Introduction law (Theorem 5.5) we can refine MD3.b to the following primitive level specifications:

$${}_{\mathsf{MinDist}.a.b}\!\vdash\ \ \circlearrowright(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n) \tag{7.13}$$
$${}_{\mathsf{MinDist}.a.b}\!\vdash\ \ \circlearrowright(\mathsf{preOk}^n \wedge \mathsf{dataOk}^n \wedge \mathsf{ok}_b^n.(r.a.b'.b)) \tag{7.14}$$
$${}_{\mathsf{MinDist}.a.b}\!\vdash\ \ (\mathsf{preOk}^n \wedge \mathsf{dataOk}^n)\ \mathsf{ensures}\ \mathsf{ok}_b^n.(r.a.b'.b) \tag{7.15}$$

The progress specification (7.15) can be simplified using (7.13):

$\Leftarrow$ { $\mathsf{MinDist} = (\llbracket a : a \in V : \mathsf{MinDist}.a)$; $\rightsquigarrow$ TRANSPARENCY }
$(\forall a : a \in V : \mathsf{true}\ {}_{\mathsf{MinDist}.a}\!\vdash \mathsf{true} \rightsquigarrow (\forall b : b \in V : d.a.b = \delta.a.b))$

So, MD0 can be refined by MD1:

For all $a \in V$:

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

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 in the program itself we do not actually have to be aware of them). Let us say that upon finishing a round n the program $\mathsf{MinDist}.a$ establishes and maintains $q.n$ where:

$$q.n \;=\; (\forall b : b \in V : \delta.a.b \leq n \Rightarrow (d.a.b = \delta.a.b)) \tag{7.8}$$

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:

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

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 7.6 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 7.7 : 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 $f \in V \rightarrow A$:

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

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

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

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.

7.5 Example: Computing Minimal Distances

Recall again the program MinDist from Section 2, which 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	$(\llbracket a : a \in V : d.a.a := 0) \rrbracket$
	$(\llbracket a, b : a, b \in V \wedge (a \neq b) : d.a.b := (\mathsf{min}(d.a * E.b)) + 1)$

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 7.6 :

$$(\forall a : a \in V : \delta.a.a = 0) \ \wedge \ (\forall a, b : a, b \in V \wedge a \neq b : \delta.a.b = (\mathsf{min}(\delta.a * E.b)) + 1)$$

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

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

Observe that upon reaching its fixpoint, the program MinDist will have the value of d satisfying the equation in Theorem 7.6, 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 that in the above program, as are the programs from the previous examples, it is assumed that each (parallel) component can directly access the variables of a neighboring component. For some architectures such a direct access is not possible and all communication has to take place through channels. Let us now try to also take this into account. We model channels with link registers. The mechanism is simple, but captures the idea of asynchronous communication adequately. Sending a messege into a channel is modelled by writting to a link register. There is no primitive synchronization: the reciever 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 it has sent, but which the reciever has not yet read. This mimics the fact that messages can be lost and hence that channels may be unreliable (however, due to the fairness assumption of UNITY, it is not possible in this model for a channel to loose messages continously). If a program can be shown to work properly using link registers then it is is tolerant to this kind of unreliability.

First of all, we observe from Theorem 7.6 that $\delta.a$ can be computed without any information about $\delta.a'$ for distinct a'. We can delegate this task to a component program.

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

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

$\Leftarrow$ { $\rightsquigarrow$ Conjunction }

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

$\Leftarrow$ { (ensures,$\mapsto$) Introduction; $\circlearrowleft$ Compositionality }
$({}_{P}\vdash \circlearrowleft \mathsf{preOk}^i) \wedge ({}_{P.i}\vdash \circlearrowleft (\mathsf{preOk}^i \wedge \mathsf{ok}^i)) \wedge ({}_{P.i}\vdash \mathsf{preOk}^i \text{ ensures } \mathsf{ok}^i)$

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 : ${}_{P.i}\vdash \mathsf{preOk}^i \text{ ensures } \mathsf{ok}^i$

In particular, M3.c states that we have to establish ok^i from preOk^i. This can be done by computing $\sqcap(x * S^*.i)$ from $\sqcap(x * S^*.j)$ of all sons j of i:

Lemma 7.5 :

$$\sqcap(x * S^*.i) \;=\; (\sqcap((\sqcap \circ (x*) \circ S^*) * S.i)) \sqcap x.i$$

Proof: To prove the above we use the following property of a semi-lattice. In a semi-lattice, the $\sqcap$ operator that belongs to that lattice satisfies:

$$\sqcap(\cup V) \;=\; \sqcap(\sqcap * V) \tag{7.7}$$

An instance of above property is: $\sqcap(U \cup V) = (\sqcap U) \sqcap (\sqcap V)$. Now let us prove the lemma above:

$\sqcap(x * S^*.i)$
$=$ { a property of S^* }
$\sqcap(x * ((\cup\{j : j \in S.i : S^*.j\}) \cup \{i\}))$
$=$ { definition $*$ }
$\sqcap(x * ((\cup(S^* * S.i)) \cup \{i\}))$
$=$ { properties of $*$: (7.6), (7.5), and (7.4) }
$\sqcap((\cup(((x*) \circ S^*) * S.i)) \cup \{x.i\})$
$=$ { (7.7) and (7.4) }
$(\sqcap((\sqcap \circ (x*) \circ S^*) * S.i)) \sqcap x.i$

■

The lemma suggests that ok^i can be established by the assignment:

$$y.i := (\sqcap((\sqcap \circ (x*) \circ S^*) * S.i)) \sqcap x.i$$

However, preOk implies that for all sons j of i, $y.j = \sqcap(x * S^*.j)$. It follows that the expression $(\sqcap \circ (x*) \circ S^*) * S.i$ in the assignment above can be replaced by $y * S.i$. So, the assignment $y.i := (\sqcap(y * S.i)) \sqcap x.i$ will do the job. Without further proof we give now a program that satisfies M3.

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

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

As its self-stabilizing property, the program automatically re-computes the value of the $y.i$'s if some adversary changes the value of x.

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 [7]. Using the map operator the problem can be stated as: compute $\sqcap(x * S^*.\alpha)$. If we let the result to be stored in $y.\alpha$, the problem can be specified as follows:

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

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

For all $i \in V$:

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

ok^i states that process i has a *'correct'* value of $y.i$ and preOk^i states that all processes that 'preceed' i, which here means being proper descendants of i, have correct values of their y's.

As a preparation for further calculation let us express M1 in terms of ok and then strengthen the goal to include a similar goal for each process i:

$$\begin{array}{ll} & \mathsf{true} \rightsquigarrow (y.\alpha = \sqcap(x * S^*.\alpha)) \\ = & \quad \{ \text{ definition ok } \} \\ & \mathsf{true} \rightsquigarrow \mathsf{ok}^\alpha \\ \Leftarrow & \quad \{ \rightsquigarrow \textsc{Substitution} \} \\ & \mathsf{true} \rightsquigarrow (\forall i : i \in V : \mathsf{ok}^i) \end{array}$$

So, M1 can be refined by M2:

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

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 ok^i using this knowledge. This is done repeatedly until ok^α is established. This sounds very much like round deomposition: 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:

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

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 $P = (\lbag i : i \in V : P.i)$ consists of programs that are pair-wise write-disjoint, which is nice because we can now apply the Transparency principle. We continue the calculation:

$$\begin{array}{ll} & \mathsf{preOk}^i\ {}_P\vdash\ \mathsf{true} \rightsquigarrow \mathsf{ok}^i \\ \Leftarrow & \quad \{ P = (\lbag j : j \in V : P.j); \rightsquigarrow \textsc{Transparency};\ \circlearrowleft \textsc{Compositionality} \} \\ & ({}_P\vdash \circlearrowleft \mathsf{preOk}^i)\ \wedge\ (\mathsf{preOk}^i\ {}_{P.i}\vdash\ \mathsf{true} \rightsquigarrow \mathsf{ok}^i) \end{array}$$

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

$\Leftarrow$ { Well-founded Induction }

$(\forall n : n \in A : (\forall m : m \prec n : J \vdash \mathsf{true} \rightsquigarrow q.m) \;\Rightarrow\; (J \vdash \mathsf{true} \rightsquigarrow q.n))$

If n is a minimal element then:

$(\forall m : m \prec n : J \vdash \mathsf{true} \rightsquigarrow q.m) \;\Rightarrow\; (J \vdash \mathsf{true} \rightsquigarrow q.n)$

$\Leftarrow$ { predicate calculus }

$J \vdash \mathsf{true} \rightsquigarrow q.n$

$=$ { n is a minimal element, hence there is no m such that $m \prec n$ }

$J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \rightsquigarrow q.n$

If n is not a minimal element then:

$(\forall m : m \prec n : J \vdash \mathsf{true} \rightsquigarrow q.m) \;\Rightarrow\; (J \vdash \mathsf{true} \rightsquigarrow q.n)$

$\Leftarrow$ { $\rightsquigarrow$ Conjunction }

$(J \vdash \mathsf{true} \rightsquigarrow (\forall m : m \prec n : q.m)) \;\Rightarrow\; (J \vdash \mathsf{true} \rightsquigarrow q.n)$

$\Leftarrow$ { $\rightsquigarrow$ Transitivity }

$J \vdash (\forall m : m \prec n : q.m) \rightsquigarrow q.n$

$\Leftarrow$ { $\rightsquigarrow$ Stable Shift }

$(\forall m : m \prec n : q.m) \in \mathsf{Pred}.(\mathbf{w}P) \;\wedge\; (\circlearrowleft J) \;\wedge\; (J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \rightsquigarrow q.n)$

$\Leftarrow$ { $\rightsquigarrow$ Confinement ; confinement is preserved by $\forall$ }

$(\circlearrowleft J) \;\wedge\; (J \wedge (\forall m : m \prec n : q.m) \vdash \mathsf{true} \rightsquigarrow q.n)$

■

In the next two subsections we will give examples illustrating the use the Round Decomposition principle. In the first example we show a derivation of a self-stabilizing program which computes the minimum input value of a set of processes connected in a tree. The derivation also demonstrates how our intuitive ideas regarding the division of tasks among component programs can be translated to the formal level using the compositionality laws from Section 6. The second example is about the program MinDist from Section 2. The self-stabilizing property of this program is not easy to be proven. In addition, we deliberately make the problem more complicated by forbidding a component program to directly access the state of other component. This should illustrate the complexity when dealing with self-stabilization in an asynchrounous system, in contrast to doing the same for a synchronous system. We will return to this point in Section 8.

7.4 Example: Self-stabilizing Computation of Minimum

We have a finite, non-empty set of vertices V connected to form a tree with root α. The connectivity in V is reflected by a function S such that for any $i \in V$, $S.i$ is the set of all sons of i. Just as in the case of a dag (see Subsection 7.1), we can define S^* and S^+. The first describes the set of all descendants of a given vertex, the second the set of all *proper* descendants. We can regard S^+ as a relation by defining $i\ S^+\ j = i \in S^+.j$. Since a tree is a special kind of dag, it follows that S^+, regarded as a relation, is well-founded. Note also that $S^*.\alpha = V$.

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.

Before we continue, let us first introduce a function *map* which will be useful in specifying the problem. For any function f and any set A, let $f * A$ (the map [6] of f on A) be defined as:

$$f * A \;=\; \{x : x \in A : f.x\} \tag{7.3}$$

Map satisfies the following properties:

$$f * (g * V) \;=\; (f \circ g) * V \tag{7.4}$$

$$f * (U \cup V) \;=\; (f * U) \cup (f * V) \tag{7.5}$$

$$f * (\cup V) \;=\; \cup((f*) * V) \tag{7.6}$$

[6] The notation is borrowed from Functional Programming

prog	R
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 in a non-deterministic way: there is no way of saying which of the initial values of the $x.i$'s will be picked as the common natural number. The self-stabilizing property of the program implies that when an adversary changes the value of some $x.i$ at any time, the program simply re-computes a new common natural number.

7.3 Monotonic Systems

A program is called monotonic if it makes progress through a sequence of totally ordered *phases* or *rounds*. Note however that by mapping program states to an empty domain of rounds every program can be considered monotonic . Therefore the notion of monotonic program is only useful if there is an objective, which is fulfilled when all rounds in the sequence have been passed. Usually to each round n we associate a predicate $q.n$ describing a goal to be established in that round.

Consider again the program MinDist from Section 2. The program computes the minimal distance between any pair of vertices (a, b) in a network (V, E). Let $\delta.a.b$ denote the minimal distance between a and b. Let n_{max} be the greatest of the minimal distances between two vertices in the network (V, N). Let $q.n$, the goal of round n, be:

$$q.n \;=\; (\forall a, b : a, b \in V : \delta.a.b \leq n \Rightarrow d.a.b = \delta.a.b) \tag{7.2}$$

If the program is indeed monotonic with respect to $[0, \ldots, n_{max}]$ 7as the domain of the rounds, ordered by $<$, then we know that eventually, when all rounds have been passed, hence also the last round, that all distances have been computed correctly.

In sequential programming, we have a law for decomposing a while-loop specification into the specifications of the loop's guard and body. Monotonic programs are comparable with while-loops. One may expect that an analogous decomposition principle holds for monotonic programs. Consider a finite set A of rounds, totally ordered by $\prec$. Suppose we want to establish $(\forall n : n \in A : q.n)$. If each $q.n$ is stable then we know that upon entering a round n the program will have established $(\forall m : m \prec n : q.m)$. The obligation of the round n is then to progress from $(\forall m : m \prec n : q.m)$ to $q.n$. Since the program is monotonic, it will traverse through all n's and eventually it will reach its final goal, namely $(\forall n : n \in A : q.n)$. This principle is called *round decomposition*: it decomposes the specification of a monotonic program into a round-wise specification. The principle is formulated below. Note that a total ordering is an instance of well-founded relations.

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

$$P : \frac{(\circlearrowleft J) \;\wedge\; (\forall n : n \in A : J \wedge (\forall m : m \prec n : q.m) \vdash \text{true} \rightsquigarrow q.n)}{J \vdash \text{true} \rightsquigarrow (\forall n : n \in A : q.n)}$$

Proof:
(Let P be a UNITY program) we derive:

$J \vdash \text{true} \rightsquigarrow (\forall n : n \in A : q.n)$
$\Leftarrow$ { $\rightsquigarrow$ Conjunction }
$(\forall n : n \in A : J \vdash \text{true} \rightsquigarrow q.n$

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

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 smaller —not necessarilly 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:

$$\mathsf{ts} \ = \ N \ - \ \mathsf{max}\{n : (n \leq N) \wedge (\forall i : i < n : x.i = x.0) : n\} \tag{7.1}$$

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 remain the same or it 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. Above strategy can be translated to the formal level. Continuing our calculation:

$$
\begin{array}{ll}
 & (x.0 = M) \wedge \neg\mathsf{ok} \rightarrowtail (x.0 < M) \\
\Leftarrow & \quad \{ \text{ Bounded Progress } \} \\
 & (\forall K : K < N : (x.0 = M) \wedge \neg\mathsf{ok} \wedge (\mathsf{ts} = K) \rightarrowtail ((x.0 = M) \wedge \neg\mathsf{ok} \wedge (\mathsf{ts} < K)) \vee (x.0 < M)) \\
\Leftarrow & \quad \{ (\text{ensures}, \rightarrowtail) \text{ Introduction } \} \\
 & (\forall K : K < N : (x.0 = M) \wedge \neg\mathsf{ok} \wedge (\mathsf{ts} = K) \ \mathsf{ensures} \ ((x.0 = M) \wedge \neg\mathsf{ok} \wedge (\mathsf{ts} < K)) \vee (x.0 < M))
\end{array}
$$

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

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

LS1.a: $\circlearrowleft \mathsf{ok}$

LS1.b: $(x.0 = M) \wedge \neg\mathsf{ok} \wedge (\mathsf{ts} = K)$ ensures $((x.0 = M) \wedge \neg\mathsf{ok} \wedge (\mathsf{ts} < K)) \vee (x.0 < M)$

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.

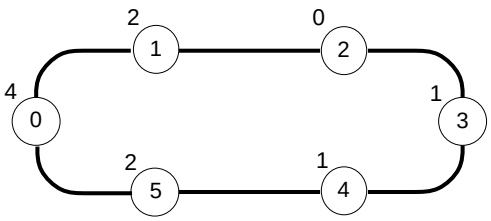

Figure 12: A ring network.

$$
\begin{array}{ll}
 & (\forall M' : M' \prec M : p \wedge (m = M') \rightsquigarrow q) \;\Rightarrow\; (p \wedge (m = M) \rightsquigarrow q) \\
\Leftarrow & \quad \{ \rightsquigarrow \text{DISJUNCTION} \} \\
 & (p \wedge (m \prec M) \rightsquigarrow q) \;\Rightarrow\; (p \wedge (m = M) \rightsquigarrow q) \\
\Leftarrow & \quad \{ q \rightsquigarrow q;\ \rightsquigarrow \text{DISJUNCTION} \} \\
 & ((p \wedge (m \prec M)) \vee q \rightsquigarrow q) \;\Rightarrow\; (p \wedge (m = M) \rightsquigarrow q) \\
\Leftarrow & \quad \{ \rightsquigarrow \text{TRANSITIVITY} \} \\
 & p \wedge (m = M) \rightsquigarrow (p \wedge (m \prec M)) \vee q
\end{array}
$$

■

As an example in the next subsection we will show a calculation for a self-stabilizing leader election.

7.2 Example: 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:

$$i^+ \;=\; (i+1) \bmod N$$

Figure 12 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 12 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 12, 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).

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:

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

Here is our strategy to solve this. 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. Let us now apply the principle to reflect the strategy. We calculate for LS0:

Theorem 7.1 : Well-founded Induction
For any well-founded relation $\prec \in A \to A \to \mathsf{bool}$:

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

For example, $<$ is well-founded. Other examples are finite dags (*directed acyclic graph*).

A dag can be represented by a pair (A, S) where A describes the set of vertices in the graph and for each $a \in A$, $S.a$ describes the set of all 'sons' of a. 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, S^+, and S^* as relations. For example: $a\ S\ b = a \in S.b$. A pair (A, S) is a dag if S^+ is irreflexive, that is, $\neg(a\ S^+\ a)$ for any $a \in A$. If (A, S) is a dag and A is finite, then S^+ is well-founded.

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-foundedness 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

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

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

Below we give the proof for the $\rightsquigarrow$ case. Notice that the proof only relies on the disjuctivity, transitivity, and reflexivity of $\rightsquigarrow$. So the principle applies to any other relation with such properties.

Proof:

$$\begin{array}{ll} & p \rightsquigarrow q \\ \Leftarrow & \quad \{\ \rightsquigarrow \text{Disjunction}\ \} \\ & (\forall M : M \in A : p \wedge (m = M) \rightsquigarrow q) \\ = & \quad \{\ \text{Well-Founded Induction}\ \} \\ & (\forall M : M \in A : (\forall M' : M' \prec M : p \wedge (m = M') \rightsquigarrow q) \;\Rightarrow\; (p \wedge (m = M) \rightsquigarrow q)) \end{array}$$

If M is a minimal element, thus there is no M' such that $M' \prec M$, we derive:

$$\begin{array}{ll} & (\forall M' : M' \prec M : p \wedge (m = M') \rightsquigarrow q) \;\Rightarrow\; (p \wedge (m = M) \rightsquigarrow q) \\ \Leftarrow & \quad \{\ \text{predicate calculus}\ \} \\ & p \wedge (m = M) \rightsquigarrow q \\ = & \quad \{\ M \text{ is a minimal element, so } (m \prec M) = \mathsf{false}\ \} \\ & p \wedge (m = M) \rightsquigarrow (p \wedge (m \prec M)) \vee q \end{array}$$

If M is not a minimal element:

$\Leftarrow$ { $\rightsquigarrow$ Transparency; $P \div Q$ }

$(J\ {}_P\!\vdash p \rightsquigarrow q) \ \wedge\ ({}_Q\!\vdash \circlearrowright J) \ \wedge\ (J\ {}_Q\!\vdash r \rightsquigarrow s) \ \wedge\ ({}_P\!\vdash \circlearrowright J)$

$=$ { $\rightsquigarrow$ Stable Background (Theorem 5.3) }

$(J\ {}_P\!\vdash p \rightsquigarrow q) \ \wedge\ (J\ {}_Q\!\vdash r \rightsquigarrow s)$

■

This concludes our discussion about compositionality. Some examples to show how we exercise formal calculation of self-stabilizing programs will be presented in the next section.

7 Examples

In this section we will present three examples of increasing complexity to illustrate how various laws which we have introduced can be used to derive self-stabilizing programs and argue about their correctness. The first example deals with the problem of leader election among a set of processes which are organized as a ring. The second one deals with the computation of the minimal input of a set of processes which are organized in a tree, and the third one is about the program MinDist introduced in Section 2. Our aim is to expose the calculation methods being used. One may compare our style of reasoning with those in [LS93, Len93]. What we would also like to illustrate is how to translate our intuitive understanding of, for example, strategies for implementing a given specification, in a natural way to the formal level.

In all the three examples we will need to use induction. Many distributed programs exploit spanning trees in their computation. Reasoning about these programs may require the use of tree induction. In sequential programming well founded induction is used to prove termination. In [CM88] Chandy and Misra use well-founded induction as a standard way to prove progress. Needless to say, induction plays an important role in programming since it enables us to reason about an unbounded number of possibilities in finitely many steps. In many textbooks inductions are formulated and treated in a semi-formal way; perhaps because the principle is so intuitive that we think we know exactly what we mean every time the word "induction" occurs in our semi-formal proofs. Sometimes however, we have to be very precise, especially when dealing with a complex problem. When such precision is required, we need not only a precise formulation of an induction principle, but also the skill and style to effectively excercise the principle at the formal level.

Often reasoning about self-stabilizing programs relies heavily on finding a well-founded relation, used to prove the progress part while preserving stability. Finding the right well-founded relation is often not easy —as in the case of, for example, the leader election and the computation of a spanning-tree [CYH91]— but in any case this is something that can be done outside the programming logic. We will begin this section by presenting the formulation of some design methods in which we typically employ well-founded induction to prove convergence. Well-founded induction is a very general induction principle. For example, it encompasses the natural number induction, tree induction, and directed acyclic graph (dag) induction.

For the sake of the readability we will omit the confinement requirement (that is, expression of the form $p \in \mathsf{Pred}.V$) from our formulas.

7.1 Bounded Progress

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. That is, an infinite sequence $\ldots \prec x_2 \prec x_1 \prec x_0$ is not possible. A well-founded relation satisfies the well-founded induction principle given below[5].

[5] It has been showed that the above formulation of well-foundedness is actually equivalent with the admittance of the well-founded induction itself.

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 theorems.

Theorem 6.8 : Spiral Law

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

Proof:

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

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

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

$\Leftarrow$ { $\rightarrowtail$ Stable Shift, Stable Background, and Confinement }

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

$\Leftarrow$ { $\rightarrowtail$ Transparency; $\circlearrowright$ Compositionality; assumptions }

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

$\Leftarrow$ { (6.1) and $\rightarrowtail$ Stable Background }

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

$\Leftarrow$ { $\rightarrowtail$ Confinement }

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

■

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, if the reader recalls, 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). See also the discussion in Section 2 and specifically Definition 2.2.

Now recall again the program MinDist from Section 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* or *non-interfering* parallel composition. For the meaning of these constructs see the discussion in Section 2, specifically Definitions 2.3 and 2.4.

Theorem 6.9 : Conjunction by $[\!]$

For any *non-empty* and *finite* set W:

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

Proof:

We are going to prove the theorem for the simple case where W consists only of two elements. The general case can be proven using finite set induction. We derive:

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

$\Leftarrow$ { $\rightsquigarrow$ Conjunction (Theorem 5.11) }

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

6.1 Write-Disjoint Composition

A much nicer compositionality rule can be obtained for a network of write-disjoint programs, that is, a network of programs which share no common write variable (Definition 2.1). In such a network a program can only write to its own variables, or to other program's input variables. Consequently, if P and Q are write-disjoint and $p \in \mathsf{Pred}.(\mathbf{w}P)$ then Q cannot destroy p. This is a crucial property in deriving the Transparency principle (2.4) we mentioned in Section 2.

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

We have defined $\rightarrowtail$ in such a way that all intermediate pairs $p' \rightarrowtail q'$ required to construct $J \ {}_{P}\vdash p \rightarrowtail q$ 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 $\circlearrowright J$ 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 6.6 : $\rightarrowtail$ Transparency

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

Proof:
By $\rightarrowtail$ Induction it suffices for us to show that $(\lambda p, q.\ J \ {}_{P[\!]Q}\vdash p \rightarrowtail q)$ is transitive, left disjunctive and includes $\mathsf{ens}.P.J$. The first two are instances of $\rightarrowtail$ Transitivity and Disjunction. The third is proven below:

$$\begin{array}{ll}
& J \ {}_{P[\!]Q}\vdash p \rightarrowtail q \\
\Leftarrow & \quad \{ \ (\mathsf{ensures}, \rightarrowtail) \ \textsc{Introduction} \ \} \\
& p, q \in \mathsf{Pred}.(\mathbf{w}(P[\!]Q)) \ \wedge \ ({}_{P[\!]Q}\vdash \circlearrowright J) \ \wedge \ ({}_{P[\!]Q}\vdash p \wedge J \ \mathsf{ensures} \ q) \\
\Leftarrow & \quad \{ \ \mathbf{w}P \subseteq \mathbf{w}P[\!]Q; \text{Confinement is monotonic (Theorem 3.2)} \ \} \\
& p, q \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ ({}_{P[\!]Q}\vdash \circlearrowright J) \ \wedge \ ({}_{P[\!]Q}\vdash p \wedge J \ \mathsf{ensures} \ q) \\
= & \quad \{ \ \circlearrowright \textsc{Compositionality} \text{ (Corollary 4.8)} \ \} \\
& p, q \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ ({}_{P}\vdash \circlearrowright J) \ \wedge \ ({}_{Q}\vdash \circlearrowright J) \ \wedge \ ({}_{P[\!]Q}\vdash p \wedge J \ \mathsf{ensures} \ q) \\
\Leftarrow & \quad \{ \ \mathsf{ensures} \ \textsc{Compositionality} \text{ (Theorem 4.9) and the definition of } \mathsf{ens} \ \} \\
& ({}_{Q}\vdash \circlearrowright J) \ \wedge \ ({}_{Q}\vdash p \wedge J \ \mathsf{unless} \ q) \ \wedge \ \mathsf{ens}.P.J.p.q \\
\Leftarrow & \quad \{ \ \mathsf{unless} \ \textsc{Post-weakening} \text{ (Theorem 4.13); definition of } \circlearrowright; \ \circlearrowright \textsc{Conjunction} \ \} \\
& ({}_{Q}\vdash \circlearrowright J) \ \wedge \ ({}_{Q}\vdash \circlearrowright p) \ \wedge \ \mathsf{ens}.P.J.p.q \\
\Leftarrow & \quad \{ \ (6.1) \ \} \\
& ({}_{Q}\vdash \circlearrowright J) \ \wedge \ p \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ (P \div Q) \ \wedge \ \mathsf{ens}.P.J.p.q \\
= & \quad \{ \text{ definition of } \mathsf{ens} \ \} \\
& ({}_{Q}\vdash \circlearrowright J) \ \wedge \ (P \div Q) \ \wedge \ \mathsf{ens}.P.J.p.q
\end{array}$$

■

An analogous law also holds for convergence.

Theorem 6.7 : $\rightsquigarrow$ Transparency

$$\frac{P \div Q \ \wedge \ ({}_{Q}\vdash \circlearrowright J) \ \wedge \ (J \ {}_{P}\vdash p \rightsquigarrow q)}{J \ {}_{P[\!]Q}\vdash p \rightsquigarrow q}$$

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

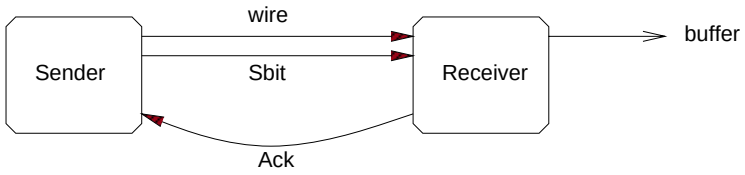

Figure 11: A simple protocol.

$$
\begin{array}{ll}
\Leftarrow & \{\ \mathsf{unless}\ \textsc{Compositionality}\ \} \\
& (_P\vdash p \wedge J\ \mathsf{unless}\ q)\ \wedge\ (_Q\vdash p \wedge J\ \mathsf{unless}\ q) \\
\Leftarrow & \{\ \mathsf{unless}\ \textsc{Post-weakening};\ \text{Definition}\ \circlearrowright\ \} \\
& (_P\vdash \circlearrowright(p \wedge J))\ \wedge\ (_Q\vdash p \wedge J\ \mathsf{unless}\ q) \\
\Leftarrow & \{\ \mathsf{unless}\ \textsc{Simple Disjunction};\ \text{Definition}\ \mathsf{unless}_V\ \} \\
& (_P\vdash \circlearrowright(p \wedge J))\ \wedge\ (_Q\vdash p \wedge J\ \mathsf{unless}_{P?!Q}\ q) \\
\Leftarrow & \{\ \rightarrowtail\ \textsc{Stable Background}\ \} \\
& (J \wedge p\ _P\vdash \mathsf{true} \rightarrowtail q)\ \wedge\ (_Q\vdash p \wedge J\ \mathsf{unless}_{P?!Q}\ q)
\end{array}
$$

■

As an example, let us consider a simple protocol as displayed in Figure 11. 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: if $\mathsf{seq}{=}\mathsf{ack}$ **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:

$$
\begin{array}{ll}
& J\ _{R[\!]S}\vdash \mathsf{true} \rightarrowtail (\mathsf{seq} = \mathsf{ack}) \\
\Leftarrow & \{\ \rightarrowtail\ \textsc{Disjunction}\ (\text{Theorem } 4.22)\ \} \\
& (\forall X :: J\ _{R[\!]S}\vdash (\mathsf{seq} = X) \rightarrowtail (\mathsf{seq} = \mathsf{ack})) \\
\Leftarrow & \{\ \text{Corollary } 6.5\ \} \\
& (\forall X :: (\ _S\vdash (\mathsf{seq} = X) \wedge J\ \mathsf{unless}_{R?!S}\ (\mathsf{seq} = \mathsf{ack}))\ \wedge\ (J \wedge (\mathsf{seq} = X)\ _R\vdash \mathsf{true} \rightarrowtail (\mathsf{seq} = \mathsf{ack})))
\end{array}
$$

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 refined to $_S\vdash (\mathsf{seq} \neq \mathsf{ack})\ \mathsf{unless}_{R?!S}\ \mathsf{false}$, which states that S cannot send anything new as long as $\mathsf{seq} \neq \mathsf{ack}$.

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.

Definition 6.1 : !?

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

Definition 6.2 : unless_V

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

Note: the dummy X has the type $\mathsf{Var} \to \mathsf{Val}$. Alternatively, by omitting some overloading we can also write the formula above as:

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

So, $P?!Q$ is the set of variables through which P reads updates made by Q. If P and Q communicates 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 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 [Pre93] is analogous to the (erronous) proof for the $\mapsto$ version of the law given in [Sin89].

Theorem 6.3 : Singh Law

$$\frac{\begin{array}{c} r, s \in \mathsf{Pred}.\mathbf{w}(P[\![Q) \;\wedge\; p_1 \in \mathsf{Pred}.(\mathbf{w}P \cup (P?!Q)) \\ (\;_{P[\![Q}\!\vdash \circlearrowleft J) \;\wedge\; (\;_Q\!\vdash r \wedge J\; \mathsf{unless}_{P?!Q}\; s) \;\wedge\; (J \wedge p_1\;\; _P\!\vdash p_2 \rightarrowtail q) \end{array}}{J\;\; _{P[\![Q}\!\vdash p_1 \wedge p_2 \wedge r \rightarrowtail q \vee \neg p_1 \vee \neg r \vee s}$$

Some corollaries of the law are for example:

Corollary 6.4 : until Compositionality

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

Corollary 6.5 :

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

Corollary 6.4 states a sufficient condition for a progress property to be preserved by the parallel composition. It is called untill Compositionality because $(J \wedge p\; \mathsf{unless}\; q) \wedge (J \vdash p \rightarrowtail q)$ corresponds with "$J \wedge p$ until q" in the linear temporal logic. Below we give the proof for Corollary 6.5. The proof of Corollary 6.4 follows more or less the same line.

Proof:

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

$q \in \mathsf{Pred}.(\mathbf{w}(P[\![Q))$ 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. As for the unless part:

$$_{P[\![Q}\!\vdash p \wedge J\; \mathsf{unless}\; q$$

Corollary 5.2 : ($\rightsquigarrow$,$\longmapsto$) Conversion

$$P, J : \frac{p \rightsquigarrow q}{p \longmapsto q}$$

Follows from the definition of $\rightsquigarrow$ and Theorem 4.29.

Corollary 5.3 : $\rightsquigarrow$ Stable Background

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

Follows from Corollaries 5.2 and 4.27.

Corollary 5.4 : $\rightsquigarrow$ Confinement

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

Follows from Corllary 5.2 and Theorem 4.28.

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

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

Follows from the definition of $\rightsquigarrow$ and Theorem 4.19.

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

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

Follows from the definition of $\rightsquigarrow$ and Corollary 4.20.

Corollary 5.7 : $\rightsquigarrow$ Substitution

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

Follows from the definition of $\rightsquigarrow$ and Theorem 4.29.

Theorem 5.8 : Accumulation

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

Theorem 5.9 : $\rightsquigarrow$ Transitivity

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

Follows from Theorem 5.8 and Corollaries 5.4 and 5.4.

Theorem 5.10 : Disjunction

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

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

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

Theorem 5.12 : $\rightsquigarrow$ Stable Shift

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

Figure 10: Some basic properties of $\rightsquigarrow$

to be replaced by arbitrary predicates. This is in fact is the definition of *convergence*, introduced by Arora and Gouda in [AG92]. Closely related concepts are the concepts of *adaptiveness* as in [GH91] and *leads-to-stabilization* as in [LS93]. There are two reasons to make this generalization. First, in some cases self-stabilization may be considered as a too strong requirement. For example we may have a system that may only start in states bounded by a predicate p. Stabilization is still possible as long as any state resulting from a sabotage by an adversary is still within p. Second, the generalized notion of stabilization enjoys many interesting properties. Our formal definition of convergence is given below.

Definition 5.1 : Convergence

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

Bear in mind that $p \rightsquigarrow q$ does not imply that q is a stable predicate. It only states that eventually q will hold forever.

We have seen an example of a $\rightsquigarrow$ specification, namely the specification of the program MinDist displayed by formula (2.1). Here are more examples. Consider a program sort whose goal is to keep an array $x[0,\ldots,n]$ sorted. Let $\mathsf{ok}.i$ mean that the initial segment $x[0,\ldots,i]$ is properly sorted, in ascending order, and that all elements in that initial segment are at most the remaining elements of the array x. Here are two stabilizing properties of sort:

$$(x = \mathsf{perm}_n.X) \ {}_{\mathsf{sort}}\!\vdash \quad \mathsf{true} \ \rightsquigarrow \ \mathsf{ok}.n \tag{5.2}$$

$$(x = \mathsf{perm}_n.X) \wedge (m < n) \wedge \mathsf{ok}.m \ {}_{\mathsf{sort}}\!\vdash \quad \mathsf{true} \ \rightsquigarrow \ \mathsf{ok}.(m+1) \tag{5.3}$$

The first states that eventually sort will stabilize to its goal. The second[4] states that once the initial segment $[x.0,\ldots,x.m]$ is ok it will remain ok and moreover the program will eventually extend the ok initial segment by one step and maintain this new situation stable.

Figure 10 displays a number of basic properties of convergence. Notice that $\rightsquigarrow$ is, in contrast to $\rightarrowtail$, not only $\vee$-junctive, but also $\wedge$-junctive. This makes $\rightsquigarrow$ calculationally attractive. The next section will present some useful compositionality properties of both operators.

6 Compositionality

The usefulness of compositionality has been motivated in Section 2. We will not repeat that discussion. It suffices here to say that a compositionality property enables us to decompose a specification of a program into specifications of its component programs. Such a property is usually expressed by a law in the form as in (2.3). In Subsection 4.3 we have seen some examples of such a property, namely the compositionality properties of unless, $\circlearrowleft$, and ensures. This section presents such properties for the progress operator $\rightarrowtail$ and the convergence operator $\rightsquigarrow$.

A general compositionality property of progress is given by the Singh law [Sin89]. 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. Before we give the formulation of the law, we will introduce some definitions.

[4] In fact, (5.3) can be obtained by applying the Round Decomposition principle (Theorem 7.4) to (5.2).

Theorem 4.25 : $\rightarrowtail$ STABLE SHIFT

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

Can be proven using $\rightarrowtail$ INDUCTION (4.7).

Theorem 4.26 : $\rightarrowtail$ STABLE STRENGTHENING

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

Can be proven using $\rightarrowtail$ INDUCTION.

Corollary 4.27 : $\rightarrowtail$ STABLE BACKGROUND

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

Follows straightforwardly from the (alternative) definition of $\rightarrowtail$ (4.6).

Theorem 4.28 : $\rightarrowtail$ CONFINEMENT

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

Can easily be proven using $\rightarrowtail$ INDUCTION.

Theorem 4.29 : $\rightarrowtail$ SUBSTITUTION

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

Follows from Corollaries 4.20 and 4.27 and Theorems 4.21 and 4.28.

Figure 9: More properties of $\rightarrowtail$

5 Stabilization

In talking about self-stabilization, we always have, perhaps implicit in our mind, besides a program that does the actual self-stabilization also an environment, which can be unstable: it may produce some transient errors, or undergo a spontaneous reconfiguration, which affects the consistency of the variables upon which the actual self-stabilization depends. Usually, this instability of the environment is abstracted by an *adversary* whose goal is to sabotage the system. In a distributed system there is also non-determinism in the order in which the actions in the system are executed. Dijkstra uses the notion of *central daemon* [Dij74] to model this. It is basically a scheduler to schedule the execution of the actions. If the schedule is non-deterministic then in a sense we can regard the central daemon as an adversary. Tied to the notion of central daemon is the notion of *privilege* [Dij74]. Only a privileged action can be elected to be executed next by the central daemon. In UNITY, its model of execution already assumes a central daemon, which is fair, but for the rest totally non-deterministic. Consequently, all actions are basically always privileged. There is however no need for us to reason about this central daemon explicitly, as the logic of UNITY already reflects its behavior.

A program P is said to self-stabilize to q if P, regardless its initial state, will establish q and maintain it. If an adversary sabotages an execution of such a program, we can regard it as if it is re-started in a new initial state. Since P can reach and maintain q regardless of its initial state, it will also do so in this new situation. So, if given enough time, such a program can always recover from any sabotage by the adversary. Note that despite the damaging behavior of the adversary, P can be designed in isolation. In Temporal Logic, if the reader is familiar with it, *"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 sight, but notice that in $P \vdash \Diamond\Box q$ the situation $\Box q$ does not have to hold on from the first time q holds but perhaps only after several iterations. The predicate q' is needed to express this possibility.

The above definition of self-stabilization can be generalized by letting the two true's in (5.1)

Theorem 4.12 : unless INTRODUCTION

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

Theorem 4.13 : unless POST-WEAKENING

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

Theorem 4.14 : SIMPLE CONJUNCTION

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

Theorem 4.15 : SIMPLE DISJUNCTION

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

Corollary 4.16 : $\circlearrowleft$ CONJUNCTION

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

Corollary 4.17 : $\circlearrowleft$ DISJUNCTION

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

Theorem 4.18 : ensures PROGRESS SAFETY PROGRESS (PSP)

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

Figure 7: Some basic laws for unless , $\circlearrowleft$, and ensures

Theorem 4.19 : (ensures,$\rightarrowtail$) INTRODUCTION

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

Corollary 4.20 : ($\Rightarrow$,$\rightarrowtail$) INTRODUCTION

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

Follows from ensures INTRODUCTION (analogous to Theorem 4.12) and Theorem 4.19

Theorem 4.21 : $\rightarrowtail$ TRANSITIVITY

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

Theorem 4.22 : $\rightarrowtail$ DISJUNCTION
For all *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 4.23 : 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 4.24 : 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 8: Properties of $\rightarrowtail$ which are analogous to those of $\mapsto$

Definition 4.11 : REACH

$$J \ {}_P\!\vdash\ p \rightarrowtail q \quad = \quad (\forall R : \mathsf{ens}.P.J \subseteq R \ \wedge \ \mathsf{Trans}.R \ \wedge \ \mathsf{Ldisj}.R : R.p.q)$$

Being the closure of ens, $J \ {}_P\!\vdash\ p \rightarrowtail q$ implies $p, q \in \mathsf{Pred}.(\mathbf{w}P)$ and ${}_P\!\vdash \circlearrowleft J$ [3]. The following alternative definition shows the latter more clearly:

$$\begin{array}{c} J \ {}_P\!\vdash\ p \rightarrowtail q \\ = \\ ({}_P\!\vdash \circlearrowleft J) \ \wedge \\ (\forall R : (\lambda r, s.\ r, s \in \mathsf{Pred}.(\mathbf{w}P) \wedge ({}_P\!\vdash (r \wedge J)\ \mathsf{ensures}\ s)) \subseteq R \wedge \mathsf{Trans}.R \wedge \mathsf{Ldisj}.R : R.p.q) \end{array} \tag{4.6}$$

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 $p, q \in \mathsf{Pred}.(\mathbf{w}P) \wedge ({}_P\!\vdash \circlearrowleft J) \wedge ({}_P\!\vdash\ J \wedge p \mapsto q)$. Note that $\rightarrowtail$ is *not* disjunctive in its J-argument. We can also define $\rightarrowtail$ as follows ($\rightarrowtail$ INDUCTION):

$$(\lambda p, q.\ J \ {}_P\!\vdash\ p \rightarrowtail q) \subseteq S \quad = \quad (\exists R : \mathsf{ens}.P.J \subseteq R \wedge \mathsf{Trans}.R \wedge \mathsf{Ldisj}.R : R \subseteq S) \tag{4.7}$$

In analogy with $\mapsto$ INDUCTION, the above definition is also called $\rightarrowtail$ INDUCTION. It can be used to inductively prove that a relation S includes $\rightarrowtail$.

As an example, consider a program buffer with $\mathbf{w}(\mathsf{buffer}) = \{\mathsf{out}\}$ and $\mathbf{i}(\mathsf{buffer}) = \{\mathsf{in}\}$. The formula:

$$(\forall X :: (\mathsf{in} = X) \ {}_{\mathsf{buffer}}\!\vdash\ \mathsf{true} \rightarrowtail (\mathsf{out} = X)) \tag{4.8}$$

states that the program buffer will eventually copy the value of in to out. However,

$$\mathsf{true} \ {}_{\mathsf{buffer}}\!\vdash\ (\mathsf{in} = X) \rightarrowtail (\mathsf{out} = X) \tag{4.9}$$

is *not* a valid expression because the argument "$\mathsf{in} = X$" is not a predicate confined by $\mathbf{w}(\mathsf{buffer})$.

As a notational convention: *if it is clear from the context which program P or which stable predicate J are meant, we often omit them from an expression.* For example we may write ${}_P\!\vdash\ p \rightarrowtail q$ or even simply $p \rightarrowtail q$ to mean $J \ {}_P\!\vdash\ p \rightarrowtail q$. Also, for laws we write, for example:

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

4.5 Basic Properties

Figure 7 displays a number of basic properties of unless, $\circlearrowleft$, and ensures taken from [CM88]. Theorems analogous to unless INTRODUCTION, POST-WEAKENING, and SIMPLE CONJUNCTION also exist for ensures. There also exist stronger CONJUNCTION and DISJUNCTION theorems for unless. See [CM88]. Corollaries 4.16 and 4.17 follow from Theorems 4.14 and 4.15.

Figures 8 displays some basic properties of $\rightarrowtail$. The proofs of these properties follow the pattern of the related proofs for $\mapsto$ properties as found in [CM88]. Figures 9 displays the properties of $\rightarrowtail$ which have no analogous $\mapsto$ properties. Note that just as in [San91] we also have the $\rightarrowtail$ SUBSTITUTION law for free.

Now how about the compositionality $\rightarrowtail$? Afterall, this was the reason why we introduced it. It suffices here to say that $\rightarrowtail$ satisfies the Transparency principle (2.4). The proof, and a further discussion on this topic, will be given later in Section 6. First we would like to discuss about how the notion of self-stabilization can be expressed and manipulated in UNITY. Afterall, self-stabilization is the topic of this paper.

[3] The reader may notice that $\rightarrowtail$ resembles the subscripted $\mapsto$ operator by Sanders [San91]. The difference is that the 'subscript', that is, the J-part, of $\rightarrowtail$ only needs to be stable whereas Sanders requires it to be an invariant. The latter results in a less compositional operator, as a parallel composition is more likely to destroy an invariant. In addition, Sanders does not require confinement by $\mathbf{w}P$.

4.3 Compositionality

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 (2.3). The usefulness of such a property has been motivated in Section 2. This subsection presents several compositionality results of the UNITY primitive operators.

The compositionality of safety properties follows a simple principle, which is interesting: the safety of a program follows from the safety of its components.

Theorem 4.7 : 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 4.8 : $\circlearrowleft$ COMPOSITIONALITY

Follows from Theorem 4.7.

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

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 4.9 : 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 The Reach Operator

Unfortunately, compositionality does not directly extend from ensures to $\mapsto$. In Section 2 we have hinted that a nice result can still be obtained if P and Q are write-disjoint ¡Definition 2.1). To refresh the reader's memory let $J \ {}_P\vdash p \mapsto q$ describe progress $p \mapsto q$ assuming that the values of the variables not writable by P $(\mathbf{w}P)^c$) satisfy J. In addition, J *fully describes* the dependency of $p \mapsto q$ on these variables. This is crucial for the transparency principle (formula (2.4) in Section 2), which states that if Q is a program which is write-disjoint with P and Q maintains J then Q cannot destroy $J \ {}_P\vdash p \mapsto q$. One way to make sure that J fully describes the dependency of $p \mapsto q$ to the variables in $(\mathbf{w}P)^c$ is to insist that the described progress is constructed from primitive progress properties which do not refer to these variables except through J. In the next subsection a new progress operator satisfying this construction scheme will be defined.

To define a progress operator that satisfies the transparency principle (2.4), first we define a variant of ensures :

Definition 4.10 : ens

$$\mathsf{ens}.P.J.p.q \quad = \quad p,q \in \mathsf{Pred}.(\mathbf{w}P) \ \wedge \ ({}_P\vdash \circlearrowleft J) \wedge ({}_P\vdash p \wedge J \text{ ensures } q)$$

The requirement $p,q \in \mathsf{Pred}.(\mathbf{w}P)$ restricts us to consider progress expressible only through the writable part of P, which is the only part of P which will ever be affected by any action of P. The intention of the stable predicate J is to capture the state of the non-writable part of P (which is of course stable in P). Note that by confining p and q by $\mathbf{w}P$ the predicate J fully describes the dependency of the described progress on $(\mathbf{w}P)^c$.

The new progress operator is called *reach*, denoted by $\rightarrowtail$. It is defined simply as the least transitive and left-disjunctive closure of ens:

Figure 6: (1) is ${}_P\vdash p$ unless q and (2) is ${}_P\vdash p$ ensures q. The thick line means a guaranteed transition.

Definition 4.4 : Left Disjunctive Relation
A relation U over $A \to \mathsf{bool}$ is called *left-disjunctive*, denoted $\mathsf{Ldisj}.U$ iff for all $q \in A \to \mathsf{bool}$ and all sets W (of predicates over A):

$$(\forall p : p \in W : U.p.q) \;\Rightarrow\; U.(\exists p : p \in W : p).q$$

Alternatively, if we write the formula without notational overloading (as warned in Section 3):

$$(\forall p : p \in W : U.p.q) \;\Rightarrow\; U.(\lambda s.\ (\exists p : p \in W : p.s)).q$$

Definition 4.5 : Leads-to

$${}_P\vdash p \mapsto q \;=\; (\forall R : (\lambda r, s.\ {}_P\vdash r \text{ ensures } s) \subseteq R \;\wedge\; \mathsf{Trans}.R \;\wedge\; \mathsf{Ldisj}.R : R.p.q)$$

Alternatively, we can write the definition of $\mapsto$ as follows:

$$(\lambda p, q.\ {}_P\vdash p \mapsto q) \;=\; \cap\{R : (\lambda r, s.\ {}_P\vdash r \text{ ensures } s) \subseteq R \wedge \mathsf{Trans}.R \wedge \mathsf{Ldisj}.R : R\} \tag{4.2}$$

which shows it more clearly that $\mapsto$ is some *least* closure of ensures. Yet another way to define $\mapsto$ is the following ($\mapsto$ Induction):

$$\begin{array}{c}(\lambda p, q.\ {}_P\vdash p \mapsto q) \subseteq S \\ = \\ (\exists R : (\lambda r, s.\ {}_P\vdash r \text{ ensures } s) \subseteq R \wedge \mathsf{Trans}.R \wedge \mathsf{Ldisj}.R : R \subseteq S)\end{array} \tag{4.3}$$

In [CM88] the definition (4.3) is called $\mapsto$ induction because it tells us how to inductively prove that a relation S includes $\mapsto$.

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

$${}_{\mathsf{MinDist}}\vdash \quad \mathsf{true} \;\mapsto\; (\forall a, b : a, b \in V : d.a.b = \delta.a.b) \tag{4.4}$$

$${}_{\mathsf{MinDist}}\vdash \quad (\forall a, b : a, b \in V : d.a.b = \delta.a.b) \;\; \mathsf{unless} \;\; \mathsf{false} \tag{4.5}$$

The first states that eventually the value of all $d.a.b$'s in the program MinDist will be equal to the actual distance from a to b. The second states that once such a situation is achieved it will remain so forever. Note that the conjunction of (4.4) and (4.5) implements the specification in (2.1). Property of the form p unless false is called *stable*. Because of its importance we will define a separate abbreviation for it.

Definition 4.6 : Stable Predicate

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

Note: if ${}_P\vdash \circlearrowleft p$ and $[\mathbf{ini}P \Rightarrow p]$ both hold then p is an *invariant*. An invariant holds throughout any execution of P. Note also that $(\lambda p.\ {}_P\vdash \circlearrowleft p)$ is neither monotonic nor anti-monotonic with respect to $\Rightarrow$.

4.1 Parallel Composition

Since there is no ordering imposed on the execution sequence of the actions, an implementator has the freedom to implement a UNITY program either as a sequential program or as a parallel program (the implementation still has to meet the fairness condition of UNITY though). Another consequence is that the parallel composition of two programs can be modelled by simply taking the union of the variables and actions of both programs. In UNITY parallel composition is denoted by $[\!]$. In [CM88] the operator is also called *program union*.

Definition 4.1 : Parallel Composition

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

Parallel composition is reflexive, commutative, and associative.

As an example the program MinDist we gave earlier can also be written using parallel composition: $\mathsf{MinDist} = ([\!]a, b : a, b \text{ in } V : \mathsf{MinDist}.a.b)$ where each program $\mathsf{MinDist}.a.b$ consists of a single action:

$$\textbf{if } a = b \textbf{ then } d.a.b := 0 \textbf{ else } d.a.b := \mathsf{min}\{b' : b' \in E.b : d.a.b' + 1\}) \tag{4.1}$$

4.2 Primitive Operators

The UNITY logic contains three primitive operators to describe the behavior of a program. These are unless, ensures, and $\mapsto$. The last was already mentioned in Section 2. It describes progress. A progress property can be broken into elementary progress properties, each guaranteed to occur by a single action. This kind of elementary progress is described by the operator ensures. The operator unless describes the safety behavior of a program. A special case of unless, namely stability, describes predicates that cannot be destroyed by a program. Just as progress, stability is required to describe a 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. Figure 6 may be helpful. $_P\vdash p$ ensures q encompasses p unless q except that there also exists an action that can, and because of the fairness assumption of UNITY, will establish q. So, unless describes safety behavior whereas ensures describes progress. As said before, ensures only describes single action progress. A more general notion of progress is obtained by taking a closure of ensures. This is the operator $\mapsto$ (read: 'leads-to'). To be more precise it is the least transitive and left-disjunctive closure of ensures. The notion of transitivity is well known; we will write $\mathsf{Trans}.R$ to denote that a relation R is transitive. The notion of left-disjunctivity is defined below. The definition of $\mapsto$ follows.

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 $(\llbracket a, b : a, b \text{ in } V : \textbf{if } \text{a=b } \textbf{then } d.a.b := 0 \textbf{ else } d.a.b := \min\{b' : b' \in E.b : d.a.b' + 1\})$

Figure 5: 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

can be simulated by a *stable predicate* (a predicate that cannot be destroyed by any action). The absence of an explicit ordering in the execution of the actions may be a little confusing at first sight, but this is simply how a parallel execution of the actions is modelled in UNITY. For the fairness condition to make sense, it is *assumed* that all (atomic) actions of a UNITY program *do not abort* and *always terminate*. Below is the syntax of a UNITY program.

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

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

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

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 is true then one is selected non-deterministically. Because an action is not allowed to abort, *a guarded assignment behaves like* **skip** *if none of its guards is true.*

In addition we have the following requirements regarding the well-formedness of a program: (1) a UNITY 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 are perfectly natural requirements for a program. Most programs that we write will satisfy them. It should be emphasized that a precise formulation of above requirement, which we will avoid in this paper, is crucial in proving compositionality laws presented in this paper. See for example [Pra93a].

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 5 displays how the program MinDist from Section 2 can be written in UNITY.

To access each component of a program we introduce the following destructors. Some of them have already been used in Section 2.

PROGRAM DESTRUCTORS:

For any UNITY program P: $\mathbf{a}P, \mathbf{r}P, \mathbf{w}P$, 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 of P. In addition, $\mathbf{i}P$ denotes the set of input variables of P, that is the variables read by, but not written by P. So, $\mathbf{i}P = \mathbf{r}P - \mathbf{w}P$.

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

Notation	Meaning
false	$(\lambda s.\ \mathsf{false})$
$p \Rightarrow q$	$(\lambda s.\ p.s \Rightarrow q.s)$
$p \vee q$	$(\lambda s.\ p.s \vee q.s)$
$(\exists i : P.i : p.i)$	$(\lambda s.\ (\exists i : P.i : p.i.s))$

Table 1: Overloading of the boolean operators.

[a] The dummy s ranges over program states.

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 any predicate operator in Table 1. So, for example, if $p, q \in \mathsf{Pred}.V$ then $p \wedge q \in \mathsf{Pred}.V$. As a rule of thumb, any predicate p is confined by $\mathsf{free}.p$, that is, the subset of Var containing the variables occurring free in p:

$$p \in \mathsf{Pred}.(\mathsf{free}.p) \tag{3.2}$$

Note however, that $\mathsf{free}.p$ is not necessarily the smallest set which confines p. For example, $\emptyset$ confines "$0 = x \vee 0 \neq x$". Another useful property is monotonicity:

Theorem 3.2 : Confinement Monotonicity

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

3.2 Binding Power

Figure 4 shows the relative binding power of the operators used in this paper. $\gg$ means *"bind stonger than"* and $|$ means *"bind as strongly as"*.

"." $\gg$ "**w**" | "**r**" | "**ini**" | "**a**" | "**i**" $\gg$ "^{c}" | "$\restriction$" $\gg$ "$\cap$" | "$\cup$" $\gg$ "$\in$" | "$\subseteq$" $\gg$ "$\neg$" $\gg$ "$\wedge$" | "$\vee$" $\gg$ "$\Rightarrow$" $\gg$ other operators $\gg$ "$=$" | "$\neq$"

Figure 4: 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 a programming logic for reasoning about safety and progress behavior of distributed programs. UNITY has a very simple view on programs and program executions. Indeed, simplicity has been the strength of UNITY. UNITY views a program as nothing more than a collection of *atomic* actions and an execution of a UNITY program is an infinite execution where at each step an action is selected and executed. There is no ordering imposed on the execution of the actions except that the implementation should guarantee *fairness* in the sense that every action must be executed infinitely often. There is no explicit notion of termination although it

If $P \triangleright Q$ holds, $P[\!]Q$ is called the layering of P and Q. P is here the lower layer and Q the upper layer. If $P \Cap Q$ holds, $P[\!]Q$ is called the fork of P and Q. Similarly if $P \| Q$ holds then $P[\!]Q$ is called the non-interfering parallel of P and Q.

It is obvious from the definitions that $\triangleright, \Cap$, and $\|$ are all stronger than $\div$.

We have mentioned several operators to express behavior of a program, in particular $\mapsto$ to express progress and $\rightsquigarrow$ to express self-stabilization. What we need now is a programming logic in which these operators can be defined. We want to know the properties of these operators and how to use them. We have mentioned induction as an important technique in proving the correctness of self-stabilizing programs. There will be examples illustrating formal excersices of this technique. We have mentioned compositionality which is an important property to support modular design. Is self-stabilization a compositional property of a program? And if it is, what are the results?

3 Notation

The *application* of a function f to x is written as $f.x$. Sometimes we also treat a function like an unary operator and hence there is no dot to separate it from its operand/argument like in the application of $\mathbf{w}$ in $\mathbf{w}(P[\!]Q)$.

The *restriction* of $f \in A \to B$ with respect to a set $S \subseteq A$, denoted by $f \restriction S$, is a function of type $A \to B$ with the following property:

$$(f \restriction S = g \restriction S) \;=\; (\forall x : x \in S : f.x = g.x) \tag{3.1}$$

The set notation used is standard except perhaps the following. *Set complement* is denoted by a superscript c like in S^c. The *powerset* of a set S —that is, the set of all subsets of S— is denoted by $\mathcal{P}(S)$. *Set abstraction* is written as $\{x : P.x : f.x\}$ instead of the usual $\{f.x | P.x\}$. We write $x, y \in S$ to abbreviate $x \in S \;\wedge\; y \in S$.

3.1 Predicates

A predicate over a set A is a function of type $A \to \mathsf{bool}$. For a predicate $p \in A \to \mathsf{bool}$, 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 $\mathsf{Var} \to \mathsf{Val}$. 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 $(\mathsf{Var} \to \mathsf{Val}) \to \mathsf{bool}$.

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 "$(\lambda s.\ (\exists i : P.i : s.(x.i) = 0))$"

This kind of symbols overloading causes usually no confusion. However, later there will be occasions where a careful distinction is called for. It will be helpful if the reader is well aware of this double meaning. To emphasize this, Table 1 shows the 'lifted' meaning of the boolean operators. In this paper we also use this kind of overloading (introducing separate notations may only confuse the reader). Whenever the meaning of a predicate expression is likely to confuse the reader we will also give the expanded expression, using λ notation.

A state-predicate p is said to be *confined* by a set of variables V, denoted by $p \in \mathsf{Pred}.V$, if p is either false everywhere or it does not restrict the value of any variable outside V. Its definition is as follows:

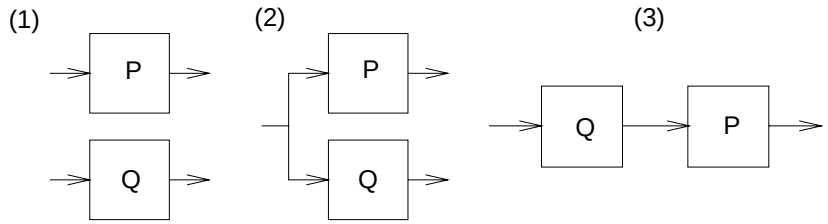

Figure 3: Instances of write-disjoint composition: (1) non-interfering composition, (2) fork, and (3) layering.

Let $P[\!]Q$ denote the parallel composition of the programs P and Q. By this we simply mean that we put P and Q side by side and execute them concurrently. How the two programs communicate is not relevant at this point. If $P \div Q$ holds, then $P[\!]Q$ is called the write-disjoint composition of P and Q.

The transparency of progress properties mentioned earlier can be expressed by the following law:

$$\frac{(J \ {}_P\!\vdash p \mapsto q) \ \wedge \ \text{"Q cannot destroy J"}}{J \ {}_{P[\!]Q}\!\vdash p \mapsto q} \tag{2.4}$$

Note that the law has the form of (2.3). A formal treatment of this kind of laws will be given in Section 6.

Composition of write-disjoint programs occurs frequently in practice (as in the program MinDist, for example). Programs that communicate through channels are also write-disjoint. Several well known constructs may be recognized as instances of write-disjoint composition. See Figure 3.

In a *non-interfering parallel composition* of two programs, all programs are independent from each other. In a *fork*, programs based their computation on the same set of input variables. For example if we have a program that computes the minimum of the values of the variables in V, and another program that computes the maximum, we can put the two programs in parallel by forking.

In a *layering* one program is called *the lower layer* the other *the upper layer*. The computation of the upper layer depends on the results of the lower layer. The converse is not necessary. For example, the lower layer can be a program that constructs a spanning tree from a vertex a and the upper layer is a program that broadcasts messages from a, using the constructed spanning tree. Layering works like a higher level sequential composition. However, the two layers do not have to be implemented sequentially, especially if they are non-terminating programs.

Let $\mathbf{r}P$, and $\mathbf{i}P$ denote respectively the set of all read and the set of all input variables of P. It is assumed that $\mathbf{w}P \subseteq \mathbf{r}P$. Note that $\mathbf{i}P = \mathbf{r}P - \mathbf{w}P$. Let us define $\triangleright$, $\Uparrow$, and $\|$ as follows.

Definition 2.2 : Layering

$$P \triangleright Q \ = \ (P \div Q) \wedge (\mathbf{w}P \supseteq \mathbf{i}Q)$$

Definition 2.3 : Fork

$$P \Uparrow Q \ = \ (P \div Q) \wedge (\mathbf{i}P = \mathbf{i}Q)$$

Definition 2.4 : Non-Interfering (Parallel)

$$P \| Q \ = \ (P \div Q) \wedge (\mathbf{r}P \cap \mathbf{r}Q = \emptyset)$$

where P and Q are programs, $\otimes$ is some kind of program composition, and spec1 and spec2 are specifications. Such properties are called *compositional*. It enables us to split the specification of $P[\!]Q$ into the specifications of P and Q. In particular, we are interested in the case where $\otimes$ is the parallel composition $[\!]$.

An important property of any program is its progress. Results on the compositionality (with respect to the parallel composition) of progress properties were however scarce. It was not until recently that significant progress was made, which we wish to bring into the reader's attention [Pra94, UHK94].

Let us now consider the following example. Let ${}_P\vdash p \mapsto q$ mean that if p holds during an execution of the program P then eventually q will hold. So, $\mapsto$ describes progress.

Let a, b, and c be boolean variables. Suppose now ${}_P\vdash a \mapsto c$ 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 ${}_P\vdash a \mapsto c$ 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**
begin
if a **then** $b :=$ true ;
if b **then** $c :=$ true
end

In this case, Q will destroy the progress $a \mapsto c$. Still, if we put P in parallel with Q' which does nothing to a and c and only writes to b under the condition, say, C then we can conclude that the composite program will have the property $a \wedge C \mapsto c \vee \neg C$ [2].

The examples suggests that recording the set of variables upon which a progress property depends may enable us to draw useful compositionality results. See for example the work by Udink, Herman, and Kok [UHK94]. 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 writable part. Compositionality can be achieved by splitting a progress specification in two parts: one to describe progress made on the write variables (and those variables only) of a program and the other part (the so-called J-part) to describe the state of the other variables. Especially interesting results can be obtained for programs that are *write-disjoint*. In this case the J-part of a program P acts as a specification for the other programs which P is composed with.

Two programs are said to be write-disjoint if their sets of write variables are disjoint. For example the programs MinDist.a.b in (2.2) are pair-wise write-disjoint.

Let $J\ {}_P\vdash p \mapsto q$ mean: (1) $p \mapsto q$ is a progress property of P, and (2) the predicate J describes the state of the variables not writable by P and in addition J also fully describes the dependency of $p \mapsto q$ on these variables. Let P and Q be two write-disjoint programs. Furthermore, P satisfies $J\ {}_P\vdash p \mapsto q$. Since P and Q are write-disjoint, Q cannot write to P's write variables. Consequently, J also fully describes the dependency of $p \mapsto q$ on Q, and hence if Q cannot destroy J neither can it destroy $p \mapsto q$. This property is called *transparency*. As we will see later, transparency turns out to be an important property.

Let $\mathbf{w}P$ denote the set of all write variables of P. Let $P \div Q$ mean that P and Q are write-disjoint:

Definition 2.1 : Write-disjoint Programs

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

[2] In fact, this is an instantiation of the Singh Law [Sin89].

```
program MinDist
init    true
begin
do forever
    for all a ∈ V do
        for all b ∈ V do
            if b = a then d.a.b := 0 else d.a.b := min{d.a.b' + 1 | b' ∈ E.b}
end
```

Figure 2: Computing minimal distances in a network.

interferes with it and tampers with the values of d, we can pretend as if the program is re-started in a new initial state. Since the program works correctly regardless its initial state, it will also do so in this new situation. Such properties are clearly very useful, and they are called *self-stabilizing* properties.

Let ${}_P\vdash p \leadsto q$ mean that if p holds somewhere during an execution of P then eventually q will hold and remain to hold forever. Let $\delta.a.b$ denote the actual minimal distance between a and b. We can use $\leadsto$ to express the self-stabilizing property of MinDist:

$$_{\mathsf{MinDist}}\vdash \mathsf{true} \leadsto (\forall a, b : a, b \text{ in } V : d.a.b = \delta.a.b) \tag{2.1}$$

The specification states that the program MinDist must eventually establish $(\forall a, b : a, b \in V : d.a.b = \delta.a.b)$. We cannot prove this dirtectly. We can however break the progress into smaller progress-steps. Induction is usually required to combine these steps into the specified progress. This is not always easy. For example naively applying an induction to the values of $d.a.b$ does not work because these values can increase or decrease during an execution[1]. Look again at the Figure 1. The number printed above a node i, $i \in \{a, b, c, d\}$, denotes the initial value of $d.a.i$. Note that the value of $d.a.a$ will decrease whereas the value of $d.a.b$ and $d.a.c$ will increase. Even an already correct value can be temporarily made incorrect. For example $d.a.d$ initially contains a correct value. However, if the process responsible for maintaining $d.a.d$ is executed first it will assign 1 to $d.a.d$, which is not the correct final value.

Indeed, induction is an important technique. In fact, many self-stabilizing programs require complicated inductive proofs (for example as in [AB89b, CYH91, Len93]). We will return to above example later in Section 7. There will also be other examples where we show how some intuitive ideas about how to (inductively) decompose a specification are translated to the formal level.

Another topic we want to address is *compositionality*. Consider again the program MinDist. We can implement it as a distributed program consisting of processes MinDist.a for all $a \in V$ where each MinDist.a maintains the variables $d.a.b$ for all $b \in V$. Even the process MinDist.a can be implemented as a distributed program consisting of processes MinDist.$a.b$, for all $b \in V$ where each process MinDist.$a.b$ does:

$$\textbf{do forever}\ \ \textbf{if}\ b = a\ \textbf{then}\ d.a.b := 0\ \textbf{else}\ d.a.b := \mathsf{min}\{d.a.b' + 1 \mid b' \in E.b\} \tag{2.2}$$

It would be nice if we could decompose a global specification into specifications of component programs. This would enable us to design each component in isolation (thus supporting the so-called *modular design approach*). In addition, this may also reduce the amount of proof obligations. To be able to do this kind of decomposition we need laws of the form:

$$\frac{(P \text{ sat spec1}) \ \wedge \ (Q \text{ sat spec2})}{P \otimes Q \text{ sat } (\text{spec1} \oplus \text{spec2})} \tag{2.3}$$

[1] Had we restricted $d.a.b$'s to initially have the value of ∞, it will be easier to prove (2.1).

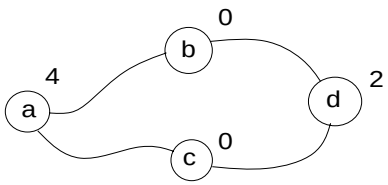

Figure 1: A simple network.

in providing solutions to the problems addressed in these examples —these are in fact well known problems— but to present a formal proof that reflects our intuitive ideas in a concise and natural way.

Although in the examples we give here we only show refinements of specifications to a certain extend —this serves our purpose here— and not a complete refinement down to the level of implementation, we do, for the sake of completeness, give an implementation for each example. It should be noted that the choice of the target architecture is a factor that cannot be ignored, even when we are still in a design phase of a self-stabilizing program. The use of channels, for example, adds extra complexity [DIM90] as the stability of the whole system in that case also depends on the stability of the channels.

In addition, the UNITY logic has been mechanically verified using a proof assistant HOL [And92, Pra93b]. The extension proposed in this paper is also mechanically verified. The proof assistant HOL is a software system, developed by M. Gordon [GM93], 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 programable proof-tools. All theorems we have verified are re-useable for further mechanical verifications. The package is available at request.

The rest of this paper is organized as follows. Section 2 provides an extensive informal motivation to the issues we wish to address in this paper. Section 3 explains the notation used in this paper. Section 4 gives a brief introduction to the programming logic UNITY and the particular extension that we use. Included are various basic laws to manipulate safety and progress specifications. Section 5 discusses how the notion self-stabilization can be formalized in UNITY and provides a set of laws to deal with it. Section 7 provides some examples in which we demonstrate the use of some of the laws. Section 8 briefly discusses some implementation aspects of self-stabilizing programs, and in Section 9 we give some conclusions.

2 An Informal Description of the Topics

This section briefly and informally explains the issues addressed in this paper. This will be helpful later when we proceed with a more rigid style of presentation as we explain the formal system that we are going to use. To avoid a too technical discussion at this early stage, some notions will be —in this section only— described less precisely. Any formal definition, either here or in any other section, is on the other hand exact.

Let us start with an example. It is about computing the *minimal distance* between any two vertices in a network. Imagine a network of vertices (a simple network is shown in Figure 1). The network consists of a set of vertices V connected to each other. The connectivity is described by a function $E \in V \to \mathcal{P}(V)$ such that $E.i$ describes the set of all neighboring vertices which are connected to the vertex i. Figure 2 displays a program that computes the minimal distances (it must be assumed that the network is *connected*).

Notice that the program has an initial condition `true` which means that it will work correctly no matter in which states it is started. Consequently, if during its execution an external agent

Formal Design of Self-stabilizing Programs

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

Rijksuniversiteit Utrecht, Vakgroep Informatica
Postbus 80.089, 3508 TB Utrecht, Nederland
Email: wishnu@cs.ruu.nl, doaitse@cs.ruu.nl

Abstract

Experience has shown that reasoning informally about distributed algorithms is extremely dangerous and error-prone, although the underlying method of reasoning is appealing. On the other hand, completely formal proofs of even simple algorithms are tedious to construct and difficult to follow. In this paper we propose a number of new operators for the UNITY logic, which enable us to reason completely formal about self-stabilizing algorithms, while maintaining the structures which play a role in the development of the algorithm. The paper includes some examples in which we show how various laws are used, and how design strategies can be represented in formal structures.

1 Introduction

The concept of *self-stabilization* was first conceived by E.W. Dijkstra [Dij74]. A self-stabilizing program is a program that will reach and remain in a set of pre-defined 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 eventually recover from any perturbation —such a perturbation may both be a failure or an update sent by the system's environment— without any outside intervention, assuming that the system is given enough time to do so. Since the work of Dijkstra many papers addressing this topic have appeared, for example [Kru79, BP89, AG92], and many self-stabilizing algorithms have been invented, for example [AB89a, AG90, CYH91, Len93].

Reasoning about self-stabilization is often complicated and it was not until recently that people attempted to deal with it more formaly. Although people are usually aware of various standard design methods, applying them formally can suddenly be an entirely different experience. Without a precise formulation one may overlook a better design course, or worse, we may reach a faulty conclusion. The idea of stabilization is first formalized by Arora and Gouda [AG90], but their reasoning about it is still done informally. A step forward is made by Herman [Her91] by proposing a number composition laws of stabilization. A truly formal treatment of stabilization is later given by Lenfert and Swierstra [LS93] who formalize the concept of stabilization in a programming logic called UNITY —designed by Chandy and Misra [CM88]— and prove various calculational properties of stabilization. In this paper we will develop the formalization in [LS93] further by adding some still missing details to it and extending it with various calculational laws, some of them, the reader may recognize, capture well-known design techniques. We also prove various compositionality results of stabilization, including the *layering principle*, an important design technique. Induction is another important technique. It is comparable with deriving the body of a loop from the specification of the loop in sequential programming. While loop refinement in sequential programming is a well formulated concept, things are less obvious in distributed programming, and thus our effort to formalize it. We will give several examples that show how to formally exercise induction to make a refinement in the way we intuitively would expect. Indeed, if a formalization is to reflect our intuition then what is an intuitively simple step should also be a simple formal step. The precision we strive for pays off as our proofs, compared to the proofs in [Len93], are much simpler and easier to follow. It should be stressed that our main concern is not

ISSN: 0924–3275

Formal Design of Self-stabilizing Programs

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

Technical Report UU-CS-1995-07
March 1995

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

Formal Design of Self-stabilizing Programs

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

UU-CS-1995-07
March 1995

Utrecht University
Department of Computer Science
Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : + 31 - 30 - 531454