Mechanically Verified Self-Stabilizing Hierarchical Algorithms

I.S.W.B. Prasetya[1*]

Fac. of Comp. Science, University of Indonesia, Depok 16424, Indonesia.
Email: wishnup@caplin.cs.ui.ac.id

Abstract. *This paper investigates self-stabilization on hierarchically divided networks. An underlying theory of self-stabilizing systems will be briefly exposed and a generic example will be given. The example and the theory have been mechanically verified using a general purpose theorem prover HOL. Three issues inherent to the problem, namely self-stabilization, concurrency, and hierarchy, can be factored out and treated one separately —something which has considerably simplified our mechanical proof (proof economy is an important issue in mechanical verification, even more than it is in the pencil and paper realm as what misleadingly appears as a few lines there may easily become a few hundreds in the mechanical world).*

1 Introduction

A self-stabilizing program is a program which is capable of converging to some pre-defined equilibrium. Such a program is tolerant to perturbations (failures, attacks) made by the its environment: if some perturbation throws it out of its equilibrium then given enough time it simply re-converges back to its equilibrium. Such a program is obviously useful. Examples include mutual exclusion protocols, communication protocols, and graph algorithms [BYC88, AG90, CYH91].

Self-stabilization is typically applied in a distributed environment. Most results deal with a flat network of processors though, and one may ask whether these results extend to hierarchical networks. Considering that many computer networks are organized hierarchically (Internet being an example thereof) the issue has practical significance. Work has been laid by Lentfert and Swierstra [LS93, Len93] who investigate self-stabilizing computation of hierarchical minimum distance in a hierarchical network. However their proof is lengthy and will only get worse if mechanized. Significant simplification is achieved by factoring out three issues inherent to the problem, namely concurrency, hierarchy, and self-stabilization, and then treating them separately. A more general approach is also taken here, resulting in a class of self-stabilization on flat networks which can be safely lifted to work on hierarchical ones.

Another issue to be highlighted here is the use of formal methods. Proving self-stabilization is very hard —see for example proof in [AB89a, AB89b, CYH91,

* The research was carried out at Utrecht University, the Netherlands.

Len93]. Formal methods were proposed both to manage the complexity and to minimize design errors. Arora and Gouda were the first who formalized the notion of self-stabilization [AG90] . However, reasoning is still carried out informally. Lenfert and Swierstra were the first who gave a truly formal framework for self-stabilizing systems [LS93]. Lenfert and Swierstra's work was later enhanced by Prasetya by introducing a more powerful operator to express stabilization in general, and providing a whole set of composition laws [Pra94b, Pra94a, Pra95]. Like all those people, we also believe in the virtue of formal methods. Through an example we will share how such a method helps us. Several most interesting laws will be shown to the readers. We urge the reader to take a look at [Pra95], where extensive lists of laws and examples of formal calculation can be found.

The results, and the underlying theory of self-stabilization have been mechanically verified using a general purpose theorem prover HOL-88. The system —made by Gordon and Melham [GM93]— is *fully expansive*, meaning that all proof strategies are built on (a small number of) axioms and primitive deduction rules, and hence there is no way one can introduce inconsistency, something which makes fully expansive theorem provers very trustworthy. A very rich collection of strategies is available, and also a meta language to let users build their own strategies. HOL is based on a higher order logic and therefore is very expressive. Each proof step is checked by HOL, but unlike model checkers, proof basically has to be hand guided. Despite this obvious disadvantage, the choice is dictated by the nature of distributed programs we seeked to investigate: (1) they consist of unbounded number of processes, and (2) they are parameterized by sophisticated structures (such as a hierarchical network). An integration of model checkers and fully expansive theorem provers will indeed be beneficial. Even now there are on-going researches heading this direction.

Due to space limitation a full calculation cannot be presented. It is also rather difficult (and less exciting) to discuss the verification in detail as it concerns thousands of lines of proof-code. So we decide not to do this. Major proof steps will however be shown, as well as some examples of proof-code and output theorems.

2 A Theory of Stabilization

This section briefly presents a theory we developed for distributed converging systems (a generalization of self-stabilizing systems). The theory is based on UNITY [CM88], a simple programming logic invented by Chandy and Misra to reason about distributed programs. A more extensive discussion on the topics can be found in [Pra95].

2.1 Programs

A program is represented by a collection of actions. Only *infinite* computations will be considered. At every step of a computation an action is *nondeterministically* selected and executed. The absence of specific orders in which

prog Fizban
read x, y, z
write x, y
init true
assign if $z = 0$ then $x := 1$ [] if $z \neq 0$ then $x := 1$ [] if $x \neq 0$ then $y, x := y + 1, 0$

Fig. 1. The program Fizban

actions are executed means that the logic does not care whether actions are to be executed sequentially or concurrently —such is considered an implementation issue by UNITY. *Weak fairness* restriction applies. All actions are assumed to be *terminating*. Guarded actions whose guards are false when executed behave as a skip. In other words, actions are *continually enabled*.

Figure 1 shows an example of a UNITY program. Our convention will be slightly different than in [CM88]. The read and write sections declare, respectively, the read and write variables of the program; the init section describes allowed initial states; the assign section describes the actions that constitute the program. Here read variables simply mean variable that can be read. *They include write variables.* Actions are listed, separated by $[]$. An action can be a simple assignment such as $x := x+1$ or a multiple assignment such as $x, y := y, x$. The meaning is as usual. An action can also be a guarded action such as:

if $x \neq 0$ then $x := 0$

Multiple guards are allowed, and if several guards evaluate to true one is selected non-deterministically. If no guard evaluates to true the action behaves like a skip. We also write, for example, $([]i : i \in V : a.i)$ which is equal to $a.i[]a.j[]a.k[]\ldots$ for all $i, j, k \in V$.

Notational convention: $a, b, c, \ldots$ range over actions; $P, Q, R, \ldots$ range over programs; x, y, z range over (program) variables; X, Y, Z range over values (of variables); and $p, q, r, \ldots$ range over state predicates. We use $\mathbf{a}P, \mathbf{ini}P, \mathbf{r}P$, and $\mathbf{w}P$ to denote, respectively, the set of actions, the initial predicate, the set of read variable, and the set of write variables of P. A UNITY program P is basically a tuple $(\mathbf{a}P, \mathbf{ini}P, \mathbf{r}P, \mathbf{w}P)$.

Parallel composition of P and Q is denoted as $P[]Q$ in UNITY. It is defined simply as the 'merge' of the component programs:

$$P[]Q \;=\; (\mathbf{a}P \cup \mathbf{a}Q, \mathbf{ini}P \wedge \mathbf{ini}Q, \mathbf{r}P \cup \mathbf{r}Q, \mathbf{w}P \cup \mathbf{w}Q) \tag{1}$$

Things can be quite subtle with parallel composition though. For example, see again program program Fizban in Figure 1; one of its property is:

$$(\forall Y :: \text{eventually } Y \leq y \text{ holds}) \tag{2}$$

If we put it in parallel with the program TikTak below, then (2) can no longer be guaranteed even though TikTak never tampers with y.

prog TikTak
assign if $z = 0$ then $z := 1$ [] if $z \neq 0$ then $z := 0$

Exercise: find out why (2) does not hold in Fizban [] TikTak.

2.2 Behavior

Stabilization consists of two aspects: *progress* and *stability*. Let us begin with stability since it is easier. A predicate p is called stable in a program P, denoted by ${}_P\vdash$ stable.p, if p is left invariant by all actions in P:

$$ {}_P\vdash \text{stable}.p \;=\; (\forall a : a \in \mathbf{a}P : \{p\}\ a\ \{p\}) \tag{3}$$

A closely related concept is *invariant*, which is a predicate that always holds through out any computation. Obviously, a stable predicate which holds initially is an invariant. However, note that not all invariants are stable!

We can generalize the concept of stable predicate by weakening it a bit. Consider a program P that when executed in p, instead of keep returning to p may (but does not have to) also step over to q. We call such behavior p unless q:

$$ {}_P\vdash p \text{ unless } q \;=\; (\forall a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{p \vee q\}) \tag{4}$$

${}_P\vdash p$ unless q does not necessarily mean that P *will* go over to q from p. However, if there exists an action a that can establish q from p, by our fairness assumption P cannot forever stay in p and hence ignoring a. Hence, a will eventually be executed and q established. This behavior is called p ensures q:

$$ {}_P\vdash p \text{ ensures } q \;=\; ({}_P\vdash p \text{ unless } q) \;\wedge\; (\exists a : a \in \mathbf{a}P : \{p \wedge \neg q\}\ a\ \{q\}) \tag{5}$$

ensures , however, only defines one-step progress, that is, progress that can be achieved through the act of one action. To include progress achieved through a cooperation of several actions, a more general operator can be defined by taking the smallest transitive and disjunctive closure of ensures . This operator is called *leads-to* in UNITY.

In this paper, a more restricted operator proposed by Prasetya will be used. The operator is written $J\ {}_P\vdash p \rightarrow q$ which is read *"P can reach q from $J \wedge p$, given the stability of J"*. p and q are also restricted to only describe values of write variables, whereas assumptions on read-only variables should now be put in J. The advantage of this operator is compositionality. Even though progress is easily destroyed by parallel composition, there are still many situations in which parallel composition does, in a sense, preserve progress made by component programs. The traditional leads-to operator is simply too liberal to be compositional in any sense and $\rightarrow$ is better in this respect. Devising a yet more compositional operator is possible, though one can expect that this is usually at the expense of simplicity.

The formal definition of $\rightarrow$ is below. A notation that will be used is $p \in \text{Pred}.V$, meaning that p is a state-predicate over variables in set V. For example $x > y$ is a state predicate over $\{x, y, z\}$, but not over $\{y, z\}$ (because it also says something about x). Roughly speaking, if V contains all free variables of p, then $p \in \text{Pred}.V$.

Definition 1. REACH OPERATOR. $\rightarrow$ is the smallest relation satisfying:

$$\frac{p, q \in \mathsf{Pred.}(\mathbf{w}P) \ \wedge \ (\ {}_P\vdash \ \mathsf{stable.}J) \ \wedge \ (\ {}_P\vdash \ J \wedge p \ \mathsf{ensures} \ q)}{J \ {}_P\vdash p \rightarrow q} \tag{6}$$

$$\frac{(J \ {}_P\vdash \ p \rightarrow q) \ \wedge \ (J \ {}_P\vdash \ q \rightarrow r)}{J \ {}_P\vdash \ p \rightarrow r} \tag{7}$$

$$\frac{(\forall p : \ p \in W : \ J \ {}_P\vdash \ p \rightarrow q)}{J \ {}_P\vdash \ (\exists p : \ p \in W : \ p) \rightarrow q}, \text{ for all non-empty } W. \tag{8}$$

For example, in program Fizban in Figure 1 we have $z = 0 \vdash \mathsf{true} \rightarrow Y < y$ and $z \neq 0 \vdash \mathsf{true} \rightarrow Y < y$ for any Y. These together express (2).

Properties for leads-to [CM88] also hold analogously for $\rightarrow$ [Pra95]. Additionally, if $J \ {}_P\vdash p \rightarrow q$ holds we know not only that P can progress from $J \wedge p$ to q, but also that J is stable and that p and q are predicates over $\mathbf{w}P$. Typically J also describes assumed values of read-only variables (note that these values are stable in P). We can also rewrite p and q using J. In the original UNITY, this ability is imposed by an axiom called Substitution Axiom [CM88]. The axiom turned out to cause inconsistency. Fortunately, we do not have this problem as our theory is purely definitional.

Exercise: find out why $\mathsf{true} \vdash \mathsf{true} \rightarrow Y < y$ cannot hold in Fizban.

Self-stabilization is defined as an ability to progress from any initial state to some pre-defined set of states (say, q) and to remain there. In temporal logic ala [MP92] this can be expressed as $\Diamond\Box q$. In our theory this is expressed as:

$$P \text{ self-stabilizes to } q \ = \ (\exists q' :: (\mathsf{true} \ {}_P\vdash \mathsf{true} \rightarrow q \wedge q') \ \wedge \ (\ {}_P\vdash \mathsf{stable.}q \wedge q'))$$

Indeed, we require P to stabilize to a stronger predicate, namely $q \wedge q'$, to express that things may not stabilize the first time q holds, but perhaps only after several iterations. It turns out to be very useful to generalize self-stabilization by allowing arbitrary predicates in the place of the two true's above.

Definition 2. CONVERGENCE

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

$J \ {}_P\vdash p \rightsquigarrow q$ is pronounced *"P converges from $J \wedge p$ to q"* and means that from $J \wedge p$ the program P will progress to states satisfying q, after-which q will continue to hold. Note that by definition it also follows that J is stable and that p and q are state-predicates over $\mathbf{w}P$. Self-stabilization to q is expressed by $\mathsf{true} \ {}_P\vdash \mathsf{true} \rightsquigarrow q$.

2.3 Properties of Convergence

Figure 2 shows a list of basic properties of convergence. A note for notation: we often drop the P or the J (or both) from formulas like $J \ {}_P\vdash p \rightarrow q$ if it is clear from the context which P or J are meant.

Theorem 3 is obvious. Theorem 4 states the J-part can be used to rewrite $p \rightsquigarrow q$. This is the analogous of Substitution Axiom [CM88] for convergence. Theorem 5 states that convergence is transitive, but Theorem 6 states a stronger sense of transitivity. It says that all intermediate predicates in a transitive trajectory will remain to hold. Theorem 7 states that convergence are both disjunctive and conjunctive. Compare this to progress operators such as leads-to or $\rightarrow$ which are only disjunctive. Theorem 8 states how part of J can be moved to the pre-condition part. Theorem 9 and 10 states how $\rightsquigarrow$ is preserved by parallel composition. Theorem 9 states that property $J \vdash p \rightsquigarrow q$ of program P will be preserved in $P[]Q$ as long as Q respect the stability of J and P and Q do not share write variables. Such a composition is called *write disjoint composition* and appears quite often in practice. For example, distributed systems that communicate through channels can be expressed as a composition of write-disjoint components [Pra95]. An instance of write-disjoint composition (called *layering*) in which one program write to the read-only variables of the other has also been recognized as an important technique in designing self-stabilizing systems [Her91, Aro92]. Had we used the leads-to we will not be able to derive Theorem 9. See [Pra95] for a complete list of compositional properties of $\rightarrow$ and $\rightsquigarrow$.

Theorem 11 expresses the well known principle of well-founded traversals: if P either decreases a metric m or stabilizes to q, it cannot decrease m forever and hence must eventually stabilize q. Theorem 12 is a corollary of Theorems 11 and 7. It states that if we can divide executions of P into (abstract) rounds A, and arrange that P converges to $q.n$ at each round n, then we know that when all rounds have been passed P has also stabilized to $(\forall n : n \in A : q.n)$.

3 Hierarchical Algorithms

In a hierarchical network nodes are grouped into domains and the domains are structured as a tree (the hierarchy). Many computer networks are structured like this (including Internet). Hierarchy is a useful abstraction mechanism: while domain-wide information may be visible, its interior may not —concealment of detail may be enforced by the system, or we may do it intentionally ourselves to simply ignore some lower level detail.

Take as an example message routing in a world-wide hierarchical network. Suppose now Carmen in Jakarta, Indonesia wants to send a message to Flips in London, UK. Each node in the network has a router and the message will be routed from one node to another according to its destination. Some routers know how to direct incoming messages to Flips, others may simply not care to know. As a solution, these other routers may simply route Carmen's message to *any* node in, say, the domain London and assume that the local network in London will further take care of its delivery to Flips. This scheme reduces the size of routing tables needed to be kept by the nodes. In addition, if the network shrinks and grows during their lifetime, which happen frequently in real life, maintaining the routing tables is cheaper (as addition or deletion of nodes invisible to a router will not affect the routing table maintained by the router).

Theorem 3.

$$\frac{p \leadsto q}{p \to q \quad \textit{and} \quad p, q \in \mathsf{Pred.}(\mathbf{w}P) \quad \textit{and} \quad {}_P\!\vdash \mathsf{stable.}J}$$

Theorem 4. SUBSTITUTION

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

Theorem 5. TRANSITIVITY

$$\frac{(p \leadsto q) \ \wedge \ (q \leadsto r)}{p \leadsto r}$$

Theorem 6. ACCUMULATION

$$\frac{(p \leadsto q) \ \wedge \ (q \leadsto r)}{p \leadsto q \wedge r}$$

Theorem 7. DISJUNCTION *and* CONJUNCTION

$$\frac{(p \leadsto q) \ \wedge \ (r \leadsto s)}{(p \vee r) \leadsto (q \vee s) \quad \textit{and} \quad (p \wedge r) \leadsto (q \wedge s)}$$

Theorem 8. STABLE SHIFT

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

Theorem 9. TRANSPARENCY

$$\frac{({}_Q\!\vdash \mathsf{stable.}J) \ \wedge \ (J \ {}_P\!\vdash p \leadsto q)}{J \ {}_{P|Q}\!\vdash p \leadsto q} \quad \textit{if } \mathbf{w}P \cap \mathbf{w}Q = \emptyset$$

Theorem 10. WRITE-DISJOINT CONJUNCTION

$$\frac{({}_P\!\vdash p \leadsto q) \ \wedge \ ({}_Q\!\vdash r \leadsto s)}{{}_{P|Q}\!\vdash (p \wedge r) \leadsto (q \wedge s)} \quad \textit{if } \mathbf{w}P \cap \mathbf{w}Q = \emptyset$$

Theorem 11. BOUNDED PROGRESS *Let $\prec$ be a well-founded relation on A and m be some metric function from program states to A:*

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

Theorem 12. ROUND DECOMPOSITION *Let A be finite and non-empty and $\prec$ be a well-founded relation on A:*

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

Fig. 2. Some basic properties of $\leadsto$.

In a hierarchical network we have a collection of nodes V connected by links, and a collection of domains $\mathcal{V}$. The connectivity will be described by a function N such that $N.a$ is the set of all neighbors of a. Neighbors are connected with *physical bidirectional links*. Each domain contains nodes; we will use $\mathbf{n}.\underline{a}$ to denote the set of all nodes inside the domain $\underline{a}$ (also called the *interior* of $\underline{a}$). In addition there is also an ordering $\lhd$ on the domains which describes the hierarchy among the domains. That is, $\underline{a} \lhd \underline{b}$ means that domain $\underline{a}$ is a *son* of domain $\underline{b}$ in the hierarchy, and conversely $\underline{b}$ is the *father* of $\underline{a}$. Fathers are considered to have a higher hierarchy than sons. The transitive and reflexive closure of $\lhd$ is written $\trianglelefteq$ and when $\underline{a} \trianglelefteq \underline{b}$ we say that $\underline{b}$ is a *super domain* of $\underline{a}$. A hierarchy relation $\lhd$ is expected to be *irreflexive* and such that $\trianglelefteq$ does not contain cycles (anti-symmetric), or in other words $\trianglelefteq$ should be a *partial order*.

A hierarchical network is therefore described by a tuple $(V, N, \mathcal{V}, \lhd, \mathbf{n})$ with the interpretation as described above.

Notational convention: $a, b, c, \ldots$ range over nodes and $\underline{a}, \underline{b}, \underline{c}, \ldots$ range over domains.

For the sake of simplicity let us consider nodes as a special kind of domains. A node, seen as a domain, contains itself as its only interior. Typically, $\lhd$ is expected to imply interior inclusion. That is, the interior of a domain must be included in the interior of a super domain:

$$\underline{a} \trianglelefteq \underline{b} \;\Rightarrow\; \mathbf{n}.\underline{a} \subseteq \mathbf{n}.\underline{b} \tag{9}$$

If so, then the hierarchy $\lhd$ fully defines the interior function $\mathbf{n}$. For a reason explained later, we will not require (9).

Recall the message routing scenario explained a few paragraphs earlier. Basically, what is being proposed is to define some *visibility* function v from nodes to domains. Sending a message m from a to b constitutes of sending m to the domain lowest in the hierarchy which contains b and which is visible from a. In other words, we are exploiting a sub-network obtained by restricting the original hierarchical network with the visibility function. Let us see first which sub-network it is.

The physical links at the node level define a network at the domain level: let $\underline{a} \in \mathcal{N}.\underline{b}$ means that there is a (bidirectional) link between domain $\underline{a}$ and $\underline{b}$. More specifically, $\underline{a} \in \mathcal{N}.\underline{b}$ means that there exists a link between some node in $\underline{a}$ and some node in $\underline{b}$:

$$\mathcal{N}.\underline{b} \;=\; \{\underline{a} \mid (\exists a, b : a \in \mathbf{n}.\underline{a} \;\wedge\; b \in \mathbf{n}.\underline{b} : a \in N.b)\} \tag{10}$$

Let $v.\underline{b}$ be the set of domains considered visible from $\underline{b}$. We can define $\mathcal{N}_v$ as follows:

$$\mathcal{N}_v.\underline{b} \;=\; \mathcal{N}.\underline{b} \cap v.\underline{b} \tag{11}$$

So, $\underline{a} \in \mathcal{N}_v.\underline{b}$ means that domain $\underline{a}$ and $\underline{b}$ are physically connected, and that domain $\underline{a}$ is visible **from** $\underline{b}$, meaning that $\underline{b}$ can be expected to know something about $\underline{a}$ (but not necessarily the other way around).

3.1 Choice of Visibility Function v

Each domain $\underline{a}$ may want to decide which other domains it wants to consider as visible. The more domains are visible to $\underline{a}$, the better informed $\underline{a}$ is, but the information will also take more space and effort to maintain. Depending on applications, there may be restrictions. For example, in a hierarchical routing program the combined interior of all visible domains from $\underline{b}$ must cover all nodes physically reachable from $\underline{b}$, or else there will be a physically reachable node which $\underline{b}$ does not know how to reach.

So, basically any relation between domains can be used as a visibility function. A suggestion due to Lentfert [Len93] is to consider super-domains and brothers (domains sharing one father) of a domain $\underline{c}$ to be visible to $\underline{c}$. Additionally, Lentfert requires the induced visibility relation to be transitive. Using this definition, if $\underline{a}$ is visible from $\underline{c}$, then either $\underline{a}$ is a super domain of $\underline{c}$ or it lies *at most* one level deeper than the shared ancestor between $\underline{a}$ and $\underline{c}$.

3.2 The Hierarchy Removed

Our distributed hierarchical algorithms will actually run on the network spanned by $\mathcal{N}_v$. Typically, we will have one process (possibly distributed) for each domain. This process repeatedly does some local computation. It cooperates with its neighbors by regularly informing them the (intermediate) results of its local computation. See the program HDP below (read, write, and init sections are omitted). Each HDP.$\underline{b}$ is the process associated with domain $\underline{b}$.

Definition 13. TEMPLATE FOR A HIERARCHICAL DISTRIBUTED PROGRAM
HDP $=$ $([]\underline{b} : \underline{b} \in \mathcal{V} :$ HDP.$\underline{b})$ where HDP.$\underline{b}$ is:

prog	HDP.$\underline{b}$
assign	do local processing, store the result in $y.\underline{b}$
	[] $([]\underline{c} : \underline{b} \in \mathcal{N}_v.\underline{c}$: inform $\underline{c}$ of the new value of $y.\underline{b})$

Notice that HDP does not directly 'see' the hierarchy-relation $\lhd$ but instead it relies on $\mathcal{N}_v$. It is true that the visibility function v is likely to depend on $\lhd$, but once v is set the hierarchy $\lhd$ is no longer relevant to the behavior of HDP and hence can be removed from our concern.

3.3 Distributing Domain-level Computation

The program HDP above is split into components, each of which does some processing associated to a domain. Such a domain-level component can be centralized in a server; or, it can be partitioned and distributed over a number of servers; or, it can simply be duplicated by the servers, which is indeed redundant but it does make a more robust system. In the latter option, consistency between duplicates need to be maintained. This paper restricts itself to this duplication scheme.

Since nodes are the only entities assumed to have computing ability, the role of servers is assumed by the nodes, though usually not all nodes are servers.

When an ordinary, non-server node needs information from a domain it is in, it requests the information from one of the domain's servers, the exact mechanism of which is considered less relevant to our main topic. For the sake of simplicity *we will pretend that all nodes in our hierarchical program are servers.* Outside the program there are non-server nodes, but how the data are distributed from server to non-server nodes is put outside the scope of this paper.

A node acting as a server for domain $\underline{a}$ does not have to act as a server for $\underline{b}$, even though $\underline{b}$ is an ancestor of $\underline{a}$. As long as each domain has at least one server it suffices for our model. It means that (9) does not necessarily hold, which is exactly why we did not insist on it. Having said all these, from this point on we do not distinguish between nodes and servers.

4 Example: Computation of Minimum Cost

Let $\mathsf{cost}.a.b$ denote the minimum cost of going from node a to b in a network. The notion is well known. For a flat network this is defined by the following equation. For all distinct nodes a and b:

$$\mathsf{cost}.a.a = \bot \tag{12}$$

$$\mathsf{cost}.a.b = \sqcap\{\mathsf{cost}.a.b' \oplus w.b'.b \mid b' \in N.b\} \tag{13}$$

where $w.a.b$ is the weight or the cost of the link between a and b. $\sqcap$ is the minimum operation, $\bot$ is the bottom element thereof. A simple instance of above is the following definition of distance, which should look more familiar:

$$\mathsf{cost}.a.a = 0 \tag{14}$$

$$\mathsf{cost}.a.b = \mathsf{min}\{\mathsf{cost}.a.b' + 1 \mid b' \in N.b\} \tag{15}$$

In a hierarchical network cost is measured between domains. The notion of cost can be expected to be more sophisticated than (14) and (15), which is why we prefer to keep the definition of minimal cost under-specified as in (12) and (13). For example Lentfert [Len93] defines the 'hierarchical' cost of a path s to be a vector v where $v.i$ is the total cost of all links in s that end up at nodes at 'depth' i. The hierarchical cost of two paths are to be compared lexicographically. This way making a walk through domains closer to the root is considered more expensive and hence will be avoided.

In the sequel, assume a hierarchical network $(V, N, \mathcal{V}, \lhd, \mathbf{n})$ with visibility function v. By (10) and (11) the tuple defines $\mathcal{N}_v$ (recall that $\mathcal{N}_v.\underline{b}$ is the set of all those domains physically connected to $\underline{b}$ and visible from $\underline{b}$). We want to construct a hierarchical self-stabilizing program, call it mCost, to compute minimal cost between pairs of domains. However, we do not want to do this for *all* pairs, but only between pairs $(\underline{a}, \underline{b})$ such that $\underline{b}$ are physically reachable *and* visible from $\underline{a}$. In other words, the cost is defined relative to the network spanned by $(\mathcal{V}, \mathcal{N}_v)$ (note that for the notion of 'reachable' to make sense, the notion of visibility needs to be transitive). The cost function looks in general like this:

$$\mathsf{cost}.\underline{a}.\underline{a} = \bot \tag{16}$$

$$\mathsf{cost}.\underline{a}.\underline{b} = \sqcap\{\mathsf{cost}.\underline{a}.\underline{b}' \oplus w.\underline{b}'.\underline{b} \mid \underline{b}' \in \mathcal{N}_v.\underline{b}\} \tag{17}$$

cost.$\underline{a}.\underline{b}$ can however be computed without any knowledge of cost.$\underline{a}'$ of other $\underline{a}'$ unequal to $\underline{a}$ and hence it can be computed separately. Without loss of generalization we will therefore only consider the problem of computing cost.$\underline{a}$ for a *given* $\underline{a}$. Let us call that given $\underline{a}$ dest (from destination).

Introduce for each server (node) b serving domain $\underline{b}$ a variable $y.\underline{b}.b$. Here is the specification of mCost:

$$\begin{array}{l} \mathsf{true}\ \vdash_{\mathsf{mCost}} \\ \quad \mathsf{true} \leadsto (\forall \underline{b}, b : \underline{\mathsf{dest}} \in \mathcal{N}_v^*.\underline{b} \wedge \underline{b} \in \mathcal{V} \wedge b \in \mathbf{n}.\underline{b} : y.\underline{b}.b = \mathsf{cost}.\underline{\mathsf{dest}}.\underline{b}) \end{array} \qquad (18)$$

where f^* is the standard reflexive and transitive closure of f (pretend for this purpose that the function f is a relation).

With the space allowed, it is unfortunately impossible to show detail how the specification (18) above can be further refined and verified. Three major refinement steps will however be shown below. They deal, separately, with three key issues, namely hierarchy, parallelism, and self-stabilization. This is done in the following way. In Subsection 3.2 it is pointed out that once the choice of the visibility function is determined, hierarchy plays no further role. Subsection 4.1 will show how self-stabilization can be broken down to round-wise stabilization. Subsection 4.2 will show how the responsibility of achieving this round-wise stabilization can easily be delegated to a component program if our hierarchical program consists of parallel components which are write-disjoint. The last two steps can subsequently be repeated until the obtained stabilization sub-goals each are simple enough to be implemented by a single component.

One may note those steps are as one should naturally expect. The fact that the original proof in [Len93] overlooked these supposedly natural steps and took instead a considerably more complicated approach warns us again about how distracting it is in formal methods to think in terms of their machinery rather than in terms of the problem at hand, especially in the absence of sufficient abstraction tools.

4.1 Round Decomposition

As our strategy, we divide any computation of mCost in rounds. Let A be the set of all those rounds. Each round n will establish for each domain $\underline{b}$ some predicate which afterwards is maintained stable. Let us call this predicate Ok.$\underline{b}.n$. In the end, when all rounds have been visited, the following will hold:

$$\mathsf{true} \vdash \mathsf{true} \leadsto (\forall n : n \in A : (\forall \underline{b} : \underline{\mathsf{dest}} \in \mathcal{N}_v^*.\underline{b} \wedge \underline{b} \in \mathcal{V} : \mathsf{Ok}.\underline{b}.n)) \qquad (19)$$

We will take the above as our new specification for mCost. By Theorem 4 (Substitution Rule) it refines (18) if:

$$(\forall n : n \in A : \mathsf{Ok}.\underline{b}.n) \;\Rightarrow\; (y.\underline{b}.b = \mathsf{cost}.\underline{\mathsf{dest}}.\underline{b}) \qquad (20)$$

for all $\underline{b} \in \mathcal{V}$ such that $\underline{\mathsf{dest}} \in \mathcal{N}_v^*.\underline{b}$ and b is a node in $\underline{b}$. (19) is also more general than (18) as the goal of each domain-level processing is now abstracted by Ok.

The round-wise specification can be obtained by applying Theorem 12 (Round Decomposition) to (19). We obtain:

$$(\forall m : m \prec n : (\forall \underline{b} : \underline{\mathsf{dest}} \in \mathcal{N}_v^*.\underline{b} \wedge \underline{b} \in \mathcal{V} : \mathsf{Ok}.\underline{b}.m)) \\ \vdash \mathsf{true} \leadsto (\forall \underline{b} : \underline{\mathsf{dest}} \in \mathcal{N}_v^*.\underline{b} \wedge \underline{b} \in \mathcal{V} : \mathsf{Ok}.\underline{b}.n)) \qquad (21)$$

for all rounds $n \in A$. The above specifies the obligation of the program mCost at each round. The relation $\prec$ on rounds is a well-founded relation, which is required by Theorem 12. The fact that well-foundedness is imposed means that any computation cannot go back to an already visited round. Assuming A is finite, convergence in finite time is guaranteed.

4.2 Distributing mCost

Let us call the part of mCost responsible for establishing $\mathsf{Ok}.\underline{b}$ as $\mathsf{mCost}.\underline{b}$. Typically the work is not done by all nodes (servers) in the network, but only by those nodes inside the concerning domain, which is $\underline{b}$. So $\mathsf{mCost}.\underline{b}$ consists of yet smaller parts, each doing node level computation. In any case, what we want is to delegate the mentioned task to the component $\mathsf{mCost}.\underline{b}$. This is possible by applying Write Disjoint Composition law in Theorem 10 to (21) and obtain:

$$(\forall m : m \prec n : (\forall \underline{b} : \underline{\mathsf{dest}} \in \mathcal{N}_v^*.\underline{b} \wedge \underline{b} \in \mathcal{V} : \mathsf{Ok}.\underline{b}.m)) \ {}_{\mathsf{mCost}.\underline{b}}\vdash \ \mathsf{true} \leadsto \mathsf{Ok}.\underline{b}.n \qquad (22)$$

for all $\underline{b}$ such that $\underline{\mathsf{dest}} \in \mathcal{N}_v^*.\underline{b}$ and $\underline{b} \in \mathcal{V}$. The law does however require that all those $\mathsf{mCost}.\underline{b}$'s are pair-wisely write-disjoint.

4.3 Instantiating Ok

Interestingly, the choice of Ok does not solely depend on the given problem, but also on the choice of communication method. The explanation is that we cannot in general guarantee self-stabilization without making sure that our communication devices do not, by design or by accident, keep emitting 'bad' values. In other words, we must guarantee that all communication devices also self-stabilize.

Below we give a choice of Ok that works for the minimum cost problem:

$$\mathsf{Ok}.\underline{b}.n = \\ (\forall b : b \in \mathbf{n}.\underline{b} : \mathsf{ok}.\underline{b}.n.(y.\underline{b}.b)) \wedge (\forall \underline{c}, c : \underline{b} \in \mathcal{N}_v.\underline{c} \wedge c \in \mathbf{n}.\underline{c} : \mathsf{ok}.\underline{b}.n.(r.\underline{c}.c.\underline{b}))$$

where each variable $y.\underline{b}.b$ maintained by node b is intended to in the end hold the value $\mathsf{cost}.\underline{\mathsf{dest}}.\underline{b}$; each variable $r.\underline{c}.c.\underline{b}$ maintained by node c is intended to mirror the value of $y.\underline{b}.b$'s in a neighboring domain $\underline{b}$; and ok is defined as follows:

$$\mathsf{ok}.\underline{b}.n.Y = (\mathsf{cost}.\underline{\mathsf{dest}}.\underline{b} \sqsubseteq n \Rightarrow (Y = \mathsf{cost}.\underline{\mathsf{dest}}.\underline{b})) \wedge (n \sqsubseteq \mathsf{cost}.\underline{\mathsf{dest}}.\underline{b} \Rightarrow n \sqsubseteq Y)$$

The variable r is introduced because a node c in domain $\underline{c}$ may not have a direct link to any node in a neighboring domain $\underline{b}$. To compute a new value of $y.\underline{c}.c$, c may need information from $\underline{b}$. So, this information will have to come from other nodes in $\underline{c}$ which do have direct links to some nodes in $\underline{b}$ (so-called *border nodes*). In our scheme these border nodes regularly broadcast their knowledge of $\underline{b}$

to the rest of $\underline{c}$. At node c this information is kept in $r.\underline{c}.c.\underline{b}$. More specifically, the broadcast is done by propagating the source information from one r to another. So, the role of the r's is to provide communication between two nodes, which as argued, cannot be ignored in Ok. Had we a flat network (all domains are actually nodes) then there is no need to broadcast and in this case indeed we can drop the second conjunct concerning r from the definition of Ok.

4.4 The Algorithm

An algorithm HDP.$\underline{b}$ satisfying (22) for the choice of Ok as given in Subsection 4.3 is given below —by the argument given earlier it follows that the composition of all HDP.$\underline{b}$'s satisfies the original specification (18) stating that it self-stabilizingly computes minimal cost.

Definition 14.

$$
\begin{array}{ll}
\text{prog} & \text{HDP}.\underline{b} \\
\text{assign} & (\llbracket b, \underline{c} : \underline{b} \in \mathcal{N}_v.\underline{c} \wedge b \in \mathbf{n}.\underline{b} \wedge b \text{ 'bordering' with } \underline{c} : \\
 & \qquad \text{broadcast value of } y.\underline{b}.b \text{ to all } r.\underline{c}.c.\underline{b},\ c \in \mathbf{n}.\underline{c}) \\
 & \llbracket \quad (\llbracket b : b \in \mathbf{n}.\underline{b} : y.\underline{b}.b := \varphi.\mathcal{N}_v.\underline{b}.(r.\underline{b}.b))
\end{array}
$$

where in our case of computing minimal cost, φ is defined as follow:

$$\varphi.M.\underline{\text{dest}}.f = \bot \tag{23}$$

$$\varphi.M.\underline{b}.f = \sqcap\{f.\underline{a} \oplus w.\underline{a}.\underline{b} \mid \underline{a} \in M.\underline{b}\} \quad \text{if } \underline{b} \neq \underline{\text{dest}} \tag{24}$$

Also, for the broadcast part to work, all interior nodes of any domain $\underline{c}$ must be physically reachable from border nodes between $\underline{c}$ and $\underline{b}$. The condition is met, for example, if the whole network is physically fully connected.

The above program is called HDP because it is actually just a more detailed formulation of the general template with the same name given in Definition 13. By using different φ's we can adapt HDP to solve other problems. Specification (22) is very important as it re-formulates the essence of the original problem should we try to solve it by round decomposition. In other words, given any hierarchical, distributed self-stabilization problem, if the solution can be computed by round-wisely approximating it, then (22) tell us what each domain-level processing must do in each round. The whole problem of hierarchical, distributed self-stabilization is now reduced to finding an Ok strong enough to imply the intended solution (in the sense as in (20)) and a φ such that (22) can be met. In fact, (22) can be sharpen to:

$$
\begin{array}{l}
\{(\forall m : m \prec n : (\forall \underline{b} : \underline{\text{dest}} \in M^*.\underline{b} \wedge \underline{b} \in \mathcal{V} : \text{Ok}.\underline{b}.m))\} \\
\qquad y.\underline{b}.b := \varphi.M.\underline{b}.(r.\underline{b}.b) \quad \{\text{Ok}.\underline{b}.n\}
\end{array} \tag{25}
$$

Notice also that with a proper instantiation the algorithm HDP also works for an ordinary, flat network. Since (25) does not in principle depend on the hierarchy of the given network, any flat network stabilization satisfying this specification will also work on hierarchical networks.

```
01 |-
02 sWF A r0 /\ TRANS r0 /\ ~(A={}) /\ FINITE A /\ CAP_PLa r1 /\ CAP_Closed r1 /\
03 (Top r1) IN (A_bc) /\ dClosed r1 A_bc /\ (e1 = Bot r1) /\ FINITE A_bc /\
04 CAP_PLa (r2:*D->*D->bool) /\ CAP_Closed r2 /\ GRAPH(V,N) /\ ~(V={}) /\ FINITE V /\
05 (!B. B IN V ==> FINITE (Nd B)) /\ (!B. B IN V ==> ~(Nd B = {})) /\
06 (!B C c. C IN V /\ B IN (N C) /\ c IN (Nd C) ==>
07     ~(MiCost (\c1. {c2 | c2 IN (Nd C) /\ c2 IN (Nn c1)}) r1 addWbc e1 (Border Nn Nd C B) c
08         = Top r1)) /\
09 (!c c'. CAP_Distr r1 r1 (addWbc c c')) /\ (!i c c'. ~(i = Top r1) ==>
10     I_r r1 i (addWbc c c' i)) /\
11 (!B n. B IN V /\ n IN A ==> Resolve N (A,r0) (Gen (V,N)) ok n B) /\
12 (!B b. B IN V /\ b IN (Nd B) ==> UNITY (gFSA V (Gen (V,N) B) d cp  B b)) /\
13 (!B C c. C IN V /\ B IN (N C) /\ c IN (Nd C) ==>
14     UNITY (ccFSA Nd Nn r1 r2 addWbc e1 d cp r B C c)) /\
15 (!B b. B IN V /\ b IN (Nd B) ==> STABLE (gFSA V (Gen (V,N) B) d cp  B b) J) /\
16 (!B C c. C IN V /\ B IN (N C) /\ c IN (Nd C) ==>
17     STABLE (ccFSA Nd Nn r1 r2 addWbc e1 d cp r B C c) J) /\
18 (!B C. C IN V /\ B IN (N C) ==> ~(Border Nn Nd C B ={})) /\
19 (!B1 b1 B2 C1 c1. ~(d B1 b1 = cp C1 c1 B2)) /\
20 (!B1 b1 B2 C1 c1 c2. ~(d B1 b1 = r C1 B2 c1 c2)) /\
21 (!B1 C1 c1 c2 B2 C2 c3. ~(r C1 B1 c1 c2 = cp C2 c3 B2)) /\
22 (!B1 b1 B2 b2. ~((B1,b1)=(B2,b2)) ==> ~(d B1 b1 = d B2 b2)) /\
23 (!B1 C1 c1 B2 C2 c2. ~((B1,C1,c1)=(B2,C2,c2)) ==> ~(cp C1 c1 B1 = cp C2 c2 B2))
24 (!B1 C1 c1 c2 B2 C2 c3 c4. ~((B1,C1,c1,c2)=(B2,C2,c3,c4)) ==>
25     ~(r C1 B1 c1 c2 = r C2 B2 c3 c4))
26  ==>
27  DAO (dFSA Nd Nn (V,N) r1 r2 addWbc e1 Gen d cp r)
28      J Nd (V,N) A ok (ccOK Nd Nn r1 e1 addWbc ok cp r) d r0
```

Fig. 3. The final theorem.

5 Verification

We can afford to be over-simplistic throughout this presentation because we simply want the issues to be easily understood by the reader. Things are very different in the mechanical verification world. Correctness is paramount there. The true complexity of the problem has to be confronted and handled with ultimate detail. Hidden assumptions or parameters may leave us unable to prove our final theorem. In many occasions the machine fails to prove things for us and hence has to be guided step by step. This can be very laborious. Often, manipulation which seems simple to an abstract human's mind turns out to be a lengthy piece of proof-code. In less than 10 pages we have conceptually explained how the program HDP works and how it can be derived (and verified). Its actual mechanical verification takes as much as 5500 lines of proof-code. The underlying logic is also verified (from UNITY to the convergence logic) and it takes 9300 lines of code. In addition to that, another 8900 lines of code are invested to provide us with various basic mathematical facts which are used (such as theories about relations and graphs) but not provided by HOL.

Let us now take a quick look at a small part of the code. The following code proves Theorem 4 ($\rightsquigarrow$ SUBSTITUTION). The theorem to be proved is listed in lines 3-6. The proof is quite simple (only two lines) and can be seen calling two other

properties of $\rightsquigarrow$, namely its monotonicity and its anti-monotonicity (at its last two arguments) with respect to $\Rightarrow$.

```
01 let CON_SUBST = prove_thm
02   ('CON_SUBST',
03     "!(Pr:^Uprog) J p q r s.
04         (WRITE Pr) CONF r /\ (WRITE Pr) CONF s /\
05         |== ((r AND J) IMP p) /\ |== ((q AND J) IMP s) /\ CON Pr J p q
06              ==> CON Pr J r s",
07    REPEAT STRIP_TAC
08    THEN IMP_RES_TAC CON_gMONO THEN IMP_RES_TAC CON_gANTIMONO) ;;
```

Verifying the logic has been relatively easy. Verifying a generic algorithm such as HDP is much more difficult. The code displayed in Figure 4 illustrates this well. Do not try to understand the code. Suffice to say that this complicated piece of work implements the two major steps mentioned in Subsections 4.1 and 4.2 (and there are still 5500 lines of code to prove other steps, major or minor, to complete the verification).

Figure 3 displays how our final theorem for HDP looks like. The program is called dFSA. Its definition is not shown, but it is more complicated than its simplification given in Definition 14 (it spans over 60 lines). Look at the number of parameters required by the program and its specification DA0 (12 and 10). Most of them are what we as human successfully keep implicit in our unmechanical reasoning. The final lines(27, 28) state that the program dFSA satisfies DA0, which basically is our initial specification (18). Line 11 expresses requirement (25). First conjunct in line 2 requires the well-foundedness of the ordering on rounds (r0). Lines 6-8 state that all participating nodes in a domain must be reachable from border nodes (otherwise data broadcasted from the border will not reach them). There are also minor conditions such as the one stated by lines 12-14, saying that all component programs are proper UNITY programs, and the one stated by lines 19-25, saying that all variables used in the programs are all distinct.

6 Conclusion

Hierarchical self-stabilizing distributed systems have been described. Especially in a large network, it is more economical that each node only keeps knowledge of other nodes which are considered 'visible' to that node. An example –a hierarchical algorithm to compute minimum distance– was given. Three major refinement steps of its initial specification were shown. Further refinement, which is not shown, is done basically by repeating those steps down to a certain level. A logic for self-stabilizing systems has also been shown. Both the logic and the example have been mechanically verified. Although the number of lines required to achieve this goal has been tremendous, and the code as displayed in Figure 4 does not look all too friendly either, people should not feel discouraged. The technology of computer aided verification is progressing steadily. More automation and better user interface can be expected in the near future.

The verified specification for HDP actually makes no reference to a minimal distance function. It simply assumes the condition (25), which essentially states

```
let dFSA_basic_decom = prove_thm
  (`dFSA_basic_decom`,
    ... <deleted> ...
    STRIP_TAC
    THEN MATCH_MP_TAC D_Round_Decom_1
    THEN ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC THENL
    [ %-- 1 --%
     REWRITE_TAC [D_preOK]
      THEN (CONV_TAC o RAND_CONV o RAND_CONV o ABS_CONV o MK_ABS_CONV) "m:*E"
      THEN CONF_TAC THEN BETA_TAC THEN CONF_TAC THENL
      [ %-- 1.1 --%
        REWRITE_TAC [D_dataOK_ADEF; D_domOK ]
        THEN (CONV_TAC o RAND_CONV o RAND_CONV o ABS_CONV o MK_ABS_CONV) "b:*Node"
        THEN CONF_TAC  THEN BETA_TAC
        THEN RULE_ASSUM_TAC BETA_RULE
        THEN RULE_ASSUM_TAC (REWRITE_RULE [DA1_DEF])
        THEN UNDISCH_ALL_TAC THEN REPEAT STRIP_TAC THEN RES_TAC
        THEN IMP_RES_TAC flatUprogs_INSERT_DELETE
        THEN POP_ASSUM SUBST1_TAC
        THEN MATCH_MP_TAC CONF_PAR THEN DISJ1_TAC
        THEN RULE_ASSUM_TAC (REWRITE_RULE [CON])
        THEN ASM_REWRITE_TAC[] ;
        %-- 1.2 --%
        REWRITE_TAC [D_comOK_ADEF]
        THEN (CONV_TAC o RAND_CONV o RAND_CONV o ABS_CONV o MK_ABS_CONV) "B:*D"
        THEN CONF_TAC  THEN BETA_TAC
        THEN RULE_ASSUM_TAC BETA_RULE
        THEN RULE_ASSUM_TAC (REWRITE_RULE [DA2_DEF])
        THEN UNDISCH_ALL_TAC THEN REPEAT STRIP_TAC THEN RES_TAC
        THEN IMP_RES_TAC flatUprogs_INSERT_DELETE
        THEN POP_ASSUM SUBST1_TAC
        THEN MATCH_MP_TAC CONF_PAR THEN DISJ1_TAC
        THEN RULE_ASSUM_TAC (REWRITE_RULE [CON])
        THEN ASM_REWRITE_TAC[] ] ;
      %-- 2 --%
      MATCH_MP_TAC D_Round_Decom2 THEN CONJ_TAC THENL
      [ %-- 2.1 --%
        MATCH_MP_TAC D_Round_Decom3 THEN ASM_REWRITE_TAC[]
        THEN REPEAT STRIP_TAC THEN RES_TAC
        THEN IMP_RES_TAC flatUprogs_INSERT_DELETE
        THEN POP_ASSUM SUBST1_TAC
        THEN MATCH_MP_TAC D_Round_Decom5
        THEN REPEAT STRIP_TAC THENL
        [ %-- 4sg --%
          FIRST_ASSUM MATCH_MP_TAC THEN ASM_REWRITE_TAC[] ;
          ASM_CASES_TAC "Pset DELETE ((gP (B:*Dom) (b:*Node)):^AD_Uprog) = {}"
          THENL [ ASM_REWRITE_TAC [flatUprogs_EMPTY]; ALL_TAC ]
          THEN DISJ1_TAC THEN MATCH_MP_TAC flatUprogs_UNITY
          THEN ASM_REWRITE_TAC [IN_DELETE; FINITE_DELETE]
          THEN REPEAT STRIP_TAC THEN RES_TAC ;
          IMP_RES_TAC flatUprogs_INSERT_DELETE
          THEN POP_ASSUM (SUBST1_TAC o SYM) THEN RES_TAC ;
          ASM_REWRITE_TAC[] ] ;
        %-- 2.2 --%
        MATCH_MP_TAC D_Round_Decom4
        THEN ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC THENL
        [ %-- 4sg --%
          MATCH_MP_TAC flatUprogs_UNITY THEN ASM_REWRITE_TAC[]
          THEN REPEAT STRIP_TAC
          THEN UNDISCH_ALL_TAC THEN REWRITE_TAC [SYM (SPEC_ALL MEMBER_NOT_EMPTY)]
          THEN REPEAT STRIP_TAC THEN RES_TAC THEN RES_TAC
          ... <deleted> ...
        ]]) ;;
```

Fig. 4. A sample code of the verification of HDP.

that each node-level computaion will bring the system closer to its final goal. This is very general condition, and also implies self-stabilization on ordinary networks, and hence it defines a class of self-stabilization which can safely be lifted to work on hierarchical networks.

References

[AB89a] 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.

[AB89b] Y. Afek and G.M. Brown. Self-stabilization of the alternating-bit protocol. In *IEEE 8th Symposiom on Reliable Distributed Systems*, October 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.

[Aro92] A. Arora. *A foundation for fault-tolerant computing.* PhD thesis, Dept. of Comp. Science, Univ. of Texas at Austin, 1992.

[BYC88] F. Bastani, I. Yen, and I. Chen. A class of inherently fault tolerant distributed programs. *IEEE Transactions on Software Engineering*, 14(1):1432–1442, 1988.

[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.

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

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

[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.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems—Specification.* Springer Verlag, 1992.

[Pra94a] I.S.W.B. Prasetya. A formal approach to design self-stabilizing programs. In E. Backer, editor, *Proceeding of Computing Science in the Netherlands 94*, pages 241–252. SION, Stichting Matematisch Centrum, 1994.

[Pra94b] 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.

[Pra95] I.S.W.B. Prasetya. *Mechanically Supported Design of Self-stabilizing Algorithms.* PhD thesis, Dept. of Comp. Science, Utrecht University, 1995.