Towards the Formal Design of Self-Stabilizing Distributed Algorithms

P.J.A. Lentfert, S.D. Swierstra

RUU-CS-92-25
August 1992

Utrecht University

Department of Computer Science

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

Towards the Formal Design of Self-Stabilizing Distributed Algorithms

P.J.A. Lentfert, S.D. Swierstra

Technical Report RUU-CS-92-25
August 1992

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

ISSN: 0024-3275

Towards the Formal Design of Self-Stabilizing Distributed Algorithms

P.J.A. Lentfert and S.D. Swierstra
Department of Computer Science, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht,
The Netherlands.

Abstract

Self-stabilizing algorithms are distributed algorithms which do not depend on a specific initial state of the system. Once started, a self-stabilizing system is guaranteed to reach a legitimate state, even in the presence of certain faults. This fault-tolerant behaviour make self-stabilizing algorithms very interesting.

However, self-stabilizing algorithms are usually highly complex. To ensure correctness, a formal proof must be given. In this article two new logical operators in UNITY are introduced. It is described how these operators can be used to formally specify the problem which a self-stabilizing algorithm is assumed to solve. Moreover, a general scheme is presented to specify self-stabilizing distributed algorithms, provided that there are no circular dependencies between different variables. This is illustrated by designing and verifying a distributed self-stabilizing algorithm for maintaining the maximum value of a possibly changing bag of integers.

1 Introduction

The concept of *self-stabilization* was first conceived by Dijkstra [6]. Since then a lot of work is published about the subject [1, 2, 3, 8, 10]. Informally, a distributed algorithm is said to be self-stabilizing if its specification does not depend on a specific initial state of the system. Such a self-stabilizing system, once started, is guaranteed to reach a correct ("legitimate") state without any intervention from outside.

An interesting property of a self-stabilizing system is that it is tolerant to certain faults, such as the crashing of processors and the subsequent recovery in arbitrary states. When the period between two faults is sufficiently long, the system is able to reach a legitimate (stable) state and to remain in such a state until the next faults occurs.

Self-stabilizing algorithms have been proposed for resetting [1, 2, 8], taking snapshots [8] and routing in dynamic networks [7, 11, 13, 15, 19, 24, 27].

Because distributed algorithms, especially self-stabilizing distributed algorithms, are usually highly complex, the design and verification of these algorithms are error-prone. It is our experience that, especially when using informal arguments to design and verify distributed algorithms, errors are easily made. To ensure correctness, a formal proof must be given. In another paper [12] we use the formalism UNITY [4] in the design and verification of a distributed self-stabilizing algorithm to maintain the maximum value of a bag of integers. Every node in the network has associated with it a single element of this bag of inputs, and each input value may change spontaneously. The algorithm, executed by each node, makes the system reach legitimate states. When the system reaches a legitimate state, the maximum value of the inputs is determined.

In this article, we introduce two new logical operators in UNITY, which may be helpful in the formal design and verification of distributed self-stabilizing algorithms. Section 3, after a short introduction in UNITY in Section 2, defines the new operators using basic UNITY operators. Section 4 explains how these operators can be used to formally specify the problem which a

self-stabilizing algorithm is assumed to solve. Moreover, it describes how to specify self-stabilizing distributed algorithms, provided that there are no circular dependencies between different variables.

Section 5 illustrates this by designing and verifying, using the new operators, a distributed algorithm for maintaining the maximum value of a possibly changing bag of integers. In this example the aforementioned circular dependency is avoided by imposing a spanning tree upon the network and letting each node maintain the maximum value of the inputs from its descendants in the spanning tree. The resulting specification consists of only two rules. In an earlier article [12], a specification of a similar algorithm (however not exactly the same) needed considerably more rules.

In Section 6 concluding remarks are given. In Appendix A, some theorems about the new operators, together with their proofs, are given.

2 UNITY

UNITY is a formalism for the development of parallel and distributed programs which was developed by Chandy and Misra [4]. Presently, research in UNITY is going on [5, 21, 22, 23]. In UNITY, after stating a problem specification, a solution strategy is specified to solve the problem (as stated in the problem specification). Next, a program is developed by stepwise refinement of specifications of the solution strategy, until the specification is restrictive enough to be translated into UNITY code. Examples of this approach can be found in [4, 26, 16, 9, 23, 25, 12]. This section presents the aspects of UNITY that are relevant to our discussion. For more information and many examples, see [4] and [22].

2.1 Programs in UNITY

A UNITY program consists of four parts:

The **declare**-section is a set of variable declarations.

The **initially**-section is a set of equations defining initial values for some or all the variables.

The **always**-section is a set of equations defining variables in terms of other variables.

The **assign**-section is a set of multiple, conditional assignment statements.

The equations in the **initially**-section as well as the statements in the **assign**-section are separated by the operator **[]**. This operator is symmetric, associative and idempotent.

Every statement is of the form

$$x_1, x_2, \ldots, x_n = e_1, e_2, \ldots, e_n \qquad \text{if } B$$

where each x_i is a variable, e_i an expression that may depend on all x_j's and B a boolean expression. To define variables by case analysis, as is common practice in mathematics, the symbol $\sim$ is introduced. For example,

$$x = \begin{cases} e_1 & \text{if } B_1 \\ e_2 & \text{if } B_2 \\ \vdots & \\ e_n & \text{if } B_n \end{cases}$$

is denoted as

$$\begin{array}{rcll} x & := & e_1 & \textbf{if } B_1 \sim \\ & & e_2 & \textbf{if } B_2 \sim \\ & & \vdots & \\ & & e_n & \textbf{if } B_n \end{array}$$

If none of the boolean expressions is true and this statement is selected for execution, the corresponding variable is left unchanged. If both B_i and B_j are true, then $e_i = e_j$ must hold. This guarantees that every assignment is deterministic.

To denote quantified statements, the following notation is used

$$\langle \mathbf{I} \textit{variable-list} : \textit{boolean-expression} :: \textit{statement-list} \rangle$$

For example, given arrays $X[0 \ldots n]$ and $Y[0 \ldots n]$ of integers,

$$\langle \mathbf{I} i : 0 \leq i \leq n :: X[i] := Y[i] \rangle$$

is equivalent to

$$X[0] := Y[0] \,\mathbf{I}\, X[1] := Y[1] \,\mathbf{I} \ldots \mathbf{I}\, X[n] := Y[n]$$

To denote quantified expressions, a similar notation is used. For example, given array $X[0 \ldots n]$ of integers,

$$\langle \mathbf{max}\, i : 0 \leq i \leq n :: X[i] \rangle$$

denotes the maximal element in $X[0 \ldots n]$. If there is no instance of the quantification, the value of the quantified expression is the unit element of the operator. In this example, it would be $-\infty$.

Likewise, $\langle \forall i :: X[i] > 0 \rangle$ holds if and only if all elements of X are positive. In this case, the range of i is understood from the given context. Moreover, any expression having a free variable, i.e. a variable that is neither bound nor a program variable, is assumed to be universally quantified over all possible values of the variable. For example,

$$\langle \forall i :: X[i] = k \Rightarrow Y[i] \geq k \rangle$$

is a shorthand for

$$\langle \forall k :: \langle \forall i :: X[i] = k \Rightarrow Y[i] \geq k \rangle \rangle.$$

The execution of a UNITY program starts in a state in which all variables have the values specified in the **initially**-section. Uninitialised variables have arbitrary initial values.

In a step of a program execution, any statement from the **assign**-section is selected and executed. The execution of each statement always terminates. A program execution consists of an infinite number of steps. During program execution each statement is selected and executed infinitely often.

The execution of a UNITY program does not terminate in the conventional sense. Instead, a UNITY program can reach a fixed point, i.e. a state where the execution of any statement leaves the state unchanged.

2.2 UNITY logic

In UNITY specifications are expressed making use of three logical binary operators: *unless*, *ensures* and $\mapsto$ ("leads-to"). The operator *unless* is used to describe safety properties of a program. Safety properties ensure that incorrect state transitions never occur. In other words, a correct program will never violate its safety properties. The other logical operators are used to describe progress properties of a program. A progress property is used to describe that a certain predicate will hold at some point in the future.

An important aspect of UNITY logic is the substitution axiom: if $(x = y)$ is an invariant of a program, then x can be replaced by y in all properties of the program. In [22], it is shown that the standard definitions of the operators *unless*, *ensures* and $\mapsto$ (as given in [4]) in combination with the substitution axiom give an unsound proof system. Modifications to the UNITY operators have been proposed that result in a sound and relatively complete proof system. An important aspect of the proposed definitions is that, for a single program, the theorems, derived for the standard UNITY operators, still hold for the newly defined operators.

In this section the definitions of the UNITY operators, as proposed by [22, 23], are given. The symbols p, q, r, I and J denote predicates. For a given program F, $F.INIT$ denotes a predicate that characterises the permitted initial states of F and $F.ASSIGN$ denotes the set of assignment statements of F.

2.2.1 unless

For a given program F, p *unless* q is formally defined as

$$p \textit{ unless } q \textit{ in } F = \langle \exists I :: (p \textit{ unless } q)_I \textit{ in } F \rangle$$

where

$$(p \textit{ unless } q)_I \textit{ in } F = \langle \forall s : s \in F.ASSIGN :: \{p \wedge \neg q \wedge I\}\ s\ \{p \vee q\} \rangle \wedge (F.INIT \Rightarrow I) \wedge (I \textit{ unless } \text{false } \textit{in } F)$$

Axiom: true *unless* false *in* F.

The operational interpretation of p *unless* q is that if p is true at some point in the execution of the program and q is not, in the next step of the execution p remains true or q becomes true. The axiom (true *unless* false *in* F) is used to prevent the definition to be circular.

$(p \textit{ unless } q)_I$ *in* F is read as "p unless q under I in F". The standard definition of (p *unless* q) (as given in [4]) is equivalent to $(p \textit{ unless } q)_{\text{true}}$. The difference in the operational interpretation of both definitions is the following. In the traditional definition p *unless* q states that once p becomes true at some point, p continues to hold at least until q does. However, in the new definition the unreachable states can be ignored when interpreting p *unless* q. Henceforth, the name of the program is omitted except where it is necessary.

Two important special cases of *unless*, *stable* and *invariant*, are given next.

$$(\textit{stable } p)_I = (p \textit{ unless } \text{false})_I$$
$$\textit{stable } p = p \textit{ unless } \text{false}$$

Once p becomes true during the program execution, it remains true. Note, however, that p is not guaranteed to become true ever.

The definition of the *invariant*-property, given some program F, is

$$(\textit{invariant } p)_I = ((F.INIT \Rightarrow p) \wedge (\textit{stable } p)_I)$$
$$\textit{invariant } p = ((F.INIT \Rightarrow p) \wedge \textit{stable } p)$$

That is, an invariant holds in every state during any execution.

Except for the union theorem, all of the standard theorems for combining *unless* properties also hold for the new definition. The union theorem can easily be adapted to hold for $(p \textit{ unless } q)_I$. However, this is beyond the scope of this article. (See [22, 23] for more details.)

Next, some results on the *unless* operator are presented. The results are given without proof. For detailed proofs, the interested reader is referred to [4]. Note that in the proofs the standard definitions of the UNITY operators are used. However, these proofs can easily be adapted to the definitions used in this article.

Reflexivity of *unless*:

$$p \textit{ unless } p$$

Antireflexivity of *unless*:

$$p \textit{ unless } \neg p$$

Consequence Weakening:

$$\frac{(p \textit{ unless } q)\ ,\ (q \Rightarrow r)}{p \textit{ unless } r}$$

Simple Conjunction:

$$\frac{(p \textit{ unless } q)\ ,\ (p' \textit{ unless } q')}{(p \wedge p') \textit{ unless } (q \vee q')}$$

Conjunction:

$$\frac{(p \textit{ unless } q)\ ,\ (p' \textit{ unless } q')}{(p \wedge p') \textit{ unless } ((p \wedge q') \vee (p' \wedge q) \vee (q \wedge q'))}$$

2.2.2 ensures

The logical operator *ensures* is, for a given program F, defined as

$$p \textit{ ensures } q \textit{ in } F = \langle \exists I \ :: \ (p \textit{ ensures } q)_I \textit{ in } F \rangle$$

where

$$(p \textit{ ensures } q)_I \textit{ in } F = (p \textit{ unless } q)_I \textit{ in } F \wedge \langle \exists s \ : \ s \in F.ASSIGN \ :: \ \{p \wedge \neg q \wedge I\} \ s \ \{q\} \rangle$$

The operational interpretation of p *ensures* q is that once p is true during the execution of the program, it remains true as long as q is false, q is guaranteed to become true eventually, and there exists at least one statement which guarantees this progress.

As in *unless*, the standard definition of p *ensures* q is equivalent to $(p \textit{ ensures } q)_{\text{true}}$. Again, the difference in (operational interpretation of) both definitions is that in the new definition the unreachable states can be ignored when interpreting p *ensures* q. All standard theorems, except the union theorem, for combining *ensures* properties also hold for the new definition.

2.2.3 leads-to

For a given program, $p \mapsto q$, is defined as

$$p \mapsto q \textit{ in } F = \langle \exists I \ :: \ (p \mapsto q)_I \textit{ in } F \rangle$$

where

$(p \mapsto q)_I$ in F holds if and only if it can be derived using a finite number of applications of the following rules:

ensures promotion:

$$\frac{(p \textit{ ensures } q)_I}{(p \mapsto q)_I}$$

Transitivity:

$$\frac{(p \mapsto q)_I \ , \ (q \mapsto r)_J}{(p \mapsto r)_{I \wedge J}}$$

Disjunction:

For any set W,

$$\frac{\langle \forall m \ : \ m \in W \ :: \ (p(m) \mapsto q)_{I(m)} \rangle}{(\langle \exists m \ : \ m \in W \ :: \ p(m) \rangle \mapsto q)_{(\wedge m \ : \ m \in W \ :: \ I(m))}}$$

In other words, if $p \mapsto q$ holds, then once p becomes true during the program execution, q is true or will become true eventually. In [17, 18, 20], the incompleteness of $\mapsto$ is discussed, and an alternative definition of $\mapsto$ is given which results in the completeness of $\mapsto$.

As before, the standard definition of $p \mapsto q$ is equivalent to $(p \mapsto q)_{\text{true}}$. Some results on the $\mapsto$ operator are presented below.

Implication:

$$\frac{p \Rightarrow q}{p \mapsto q}$$

Precondition Strengthening:

$$\frac{(p \Rightarrow q) \ , \ (q \mapsto r)}{p \mapsto r}$$

Consequence Weakening:

$$\frac{(p \mapsto q) \ , \ (q \Rightarrow r)}{p \mapsto r}$$

PSP (Progress-Safety-Progress):

$$\frac{(p \mapsto q)\ ,\ (r\ \textit{unless}\ b)}{(p \wedge r) \mapsto ((q \wedge r)\ \vee\ b)}$$

Cancellation:

$$\frac{(p \mapsto q \vee b)\ ,\ (b \mapsto r)}{p \mapsto q \vee r}$$

Completion:

For any finite set of predicates p_i, q_i:

$$\frac{\langle \forall i\ ::\ p_i \mapsto q_i \vee b\rangle\ ,\ \langle \forall i\ ::\ q_i\ \textit{unless}\ b\rangle}{\langle \forall i\ ::\ p_i\rangle \mapsto (\langle \forall i\ ::\ q_i\rangle \vee b)}$$

Generalization of the Completion theorem:

For any finite set of predicates p_i, q_i, r_i, b_i:

$$\frac{\langle \forall i\ ::\ p_i \mapsto q_i \vee r_i\rangle\ ,\ \langle \forall i\ ::\ q_i\ \textit{unless}\ b_i\rangle}{\langle \forall i\ ::\ p_i\rangle \mapsto (\langle \forall i\ ::\ q_i\rangle\ \vee\ \langle \exists i\ ::\ r_i \vee b_i\rangle)}$$

2.2.4 Substitution Axiom

In [23], substitution rules are given that can replace the substitution axiom (as given in [4]). Below, all occurrences of Prop can be replaced by *unless*, *ensures* or $\mapsto$. Furthermore, z and y denote either both variables or predicates, and P and Q map z and y to predicates. Then, for a single program, the following theorem holds:

Substitution:

$$\frac{(P.z\ \text{Prop}\ Q.z)\ ,\ (\textit{invariant}\ r)\ ,\ (r \Rightarrow (z = y))}{P.y\ \text{Prop}\ Q.y}$$

3 Additional Operators

This section defines two additional logical operators in UNITY, and gives some theorems about them. As will be shown in the next section, these operators are useful in the formal design and verification of (distributed) self-stabilizing algorithms. Both operators are defined in terms of the basic operators as given in Section 2. Each newly defined operator consists of two parts, a progress and a safety property.

3.1 Assures

For a given program F the logical operator *assures* is defined as

$$r\ \textit{assures}_p\ q\ \textit{in}\ F = \langle \exists I\ ::\ (r\ \textit{assures}_p\ q)_I\ \textit{in}\ F\rangle$$

where

$$(r\ \textit{assures}_p\ q)_I\ \textit{in}\ F = ((r\ \textit{ensures}\ q)_I\ \textit{in}\ F\ \wedge\ (p \wedge q \wedge r\ \textit{unless}\ \neg p)_I\ \textit{in}\ F)$$

The operational interpretation of $r\ \textit{assures}_p\ q$ is that once r is true during the execution of the program, it remains true as long as q is false, q is guaranteed to eventually become true, and there exists at least one statement which is guaranteed to cause this progress. And during the execution when p, q and r hold, these continue to hold until p does not hold anymore.

3.2 Eventually-implies

For a given program F the logical operator $\mapsto\!\!\!\!\!\!\rightarrow$, read as *eventually-implies*, is defined as

$$p \mapsto\!\!\!\!\!\!\rightarrow q \ \textit{in}\ F = \langle \exists I \ ::\ (p \mapsto\!\!\!\!\!\!\rightarrow q)_I \ \textit{in}\ F \rangle$$

where

$$(p \mapsto\!\!\!\!\!\!\rightarrow q)_I \ \textit{in}\ F = ((p \mapsto (q \vee \neg p))_I \ \textit{in}\ F \ \wedge \ (p \wedge q \ \textit{unless}\ \neg p)_I \ \textit{in}\ F)$$

The operational interpretation of $p \mapsto\!\!\!\!\!\!\rightarrow q$ is that once p holds, q is or will be true or eventually p will not hold anymore, and when p and q both hold, these continue to hold until p does not hold anymore. In other words, when p holds, within finite time $p \Rightarrow q$ and this remains to hold until p is falsified.

3.3 Theorems

In this section some theorems are stated for combining the newly introduced operators. Proofs of these theorems can be found in Appendix A.

assures promotion:

$$\frac{r \ \textit{assures}_p \ q}{(p \wedge r) \mapsto\!\!\!\!\!\!\rightarrow q}$$

Implication:

$$\frac{p \Rightarrow q}{p \mapsto\!\!\!\!\!\!\rightarrow q}$$

Reflexivity:

$$p \mapsto\!\!\!\!\!\!\rightarrow p$$

Weak Transitivity:

$$\frac{(p \mapsto\!\!\!\!\!\!\rightarrow q)\ ,\ (q \mapsto\!\!\!\!\!\!\rightarrow r)}{p \mapsto\!\!\!\!\!\!\rightarrow (q \wedge r)}$$

Restricted Consequence Weakening:

$$\frac{(p \mapsto\!\!\!\!\!\!\rightarrow (q \wedge r))\ ,\ (p \Rightarrow r)}{p \mapsto\!\!\!\!\!\!\rightarrow q}$$

Precondition Strengthening:

$$\frac{(p \Rightarrow q)\ ,\ (q \mapsto\!\!\!\!\!\!\rightarrow r)}{p \mapsto\!\!\!\!\!\!\rightarrow r}$$

Conjunction:

$$\frac{(p \mapsto\!\!\!\!\!\!\rightarrow q)\ ,\ (p' \mapsto\!\!\!\!\!\!\rightarrow q')}{(p \wedge p') \mapsto\!\!\!\!\!\!\rightarrow (q \wedge q')}$$

Generalization of the Conjunction theorem:
For any finite set of predicates p_i, q_i:

$$\frac{\langle \forall i \ ::\ p_i \mapsto\!\!\!\!\!\!\rightarrow q_i \rangle}{\langle \forall i \ ::\ p_i \rangle \mapsto\!\!\!\!\!\!\rightarrow \langle \forall i \ ::\ q_i \rangle}$$

Stability:

$$\frac{(p \mapsto\!\!\!\!\!\!\rightarrow q)\ ,\ (\textit{stable}\ p)}{(p \mapsto q), (\textit{stable}\ (p \wedge q))}$$

4 Specifications of Self-Stabilizing Algorithms in UNITY

This section describes how self-stabilizing algorithms can be specified in UNITY, using the operators as defined in Section 3. Section 4.1 describes how the problem specifications of self-stabilizing algorithms can be formulated.

Section 4.2 gives a general scheme for specifying solution strategies in UNITY which assume (a network of) nodes with the following property:

> If the state of node a depends on the state of node b ($b \neq a$), the state of b does not depend on the state of a.

4.1 Problem Specifications

In the following, the environment (env) is defined as that part of the system's state upon which the algorithm depends but which cannot be influenced (by the algorithm). For example, in the case of a routing algorithm, env denotes the topology of the system, which can be expressed as the weights of the links between every two nodes.

LS is a predicate to denote that the program state is in a legitimate state. For example, in case of the routing algorithm, LS may describe that all nodes have correct routing tables.

An obvious way to specify a self-stabilizing algorithm, in terms of temporal logic [14], is

$$(\Diamond\Box(env = v)) \Rightarrow (\Diamond\Box LS)$$

In other words, if eventually the environment is stable (does not change anymore) and has some value v, within finite time the system reaches a legitimate state, and from then on every successor of a legitimate state is legitimate.

One advantage of describing self-stabilizing algorithms this way is that properties are straightforwardly specified. Another advantage is the ease of calculating. For example, transitivity may be very helpful, i.e. the following property can be used: given $((\Diamond\Box p) \Rightarrow (\Diamond\Box q))$ and $((\Diamond\Box q) \Rightarrow (\Diamond\Box r))$, it directly follows that $((\Diamond\Box p) \Rightarrow (\Diamond\Box r))$.

Until now it has not been possible to directly translate the displayed formula (in temporal logic) into UNITY properties. An alternative in UNITY is to use $\mapsto$ (*eventually-implies*). Suppose $((env = v) \mapsto LS)$ and *stable* $(env = v)$ hold, then the stability theorem states that within finite time after env has obtained the value v, $((env = v) \wedge LS)$ will hold forever.

A disadvantage of the *eventually-implies* operator is that it is not transitive, i.e. given $p \mapsto q$ and $q \mapsto r$, it is not necessarily true that $p \mapsto r$ holds. Given $p \mapsto q$ and $q \mapsto r$, then $p \hookrightarrow (q \vee \neg p)$ does hold. However, given $p \mapsto q$ and $q \mapsto r$, it is not necessarily true that $(p \wedge q)$ *unless* $\neg p$ holds. As an example, consider the following specification of a self-stabilizing distributed algorithm for maintaining the maximum value of a frequently changing bag of inputs.

$$(env = v) \mapsto term$$

$$term \mapsto max$$

where $term$ is a predicate that describes that the system has terminated, that is the system has finished its computation, and max is a predicate that describes that the maximum value is computed. The first rule implies that once the system has terminated, this remains to hold until the environment (an input value) changes. The second rule states that once the system has terminated and the maximum value is computed, this remains to hold until the system is not terminated anymore. However, it is not necessarily true that once the maximum value is computed (and the system has not terminated yet), this remains to hold as long as the environment does not change.

Another disadvantage, compared to the rules given in temporal logic, is that generally the resulting rules are more complex. Suppose the following specification is used to specify a self-stabilizing algorithm:

$$(env = v) \mapsto LS.$$

This rule states:

- always within finite time a legitimate state is reached or the environment changes, and
- once a legitimate state has been reached, the successor states remain legitimate until the environment changes.

Opposed to the specification given in temporal logic (and opposed to our informal requirements) the following scenario is not allowed given this rule. When the environment is stable, during the computation *temporarily* a legitimate state is reached (more or less accidentally) but the system is not stable yet. Within finite time, however, after the environment has become stable the system will stabilize, i.e. a legitimate state will be reached and the state will remain legitimate. As an example of this scenario, consider again the specification of a self-stabilizing distributed algorithm for maintaining the maximum value of a frequently changing bag of inputs (as we do in Section 5). During the distributed computation of the maximum value, the computed value may temporarily be the maximum value before the system stabilizes. To allow for this behaviour, the following (general) specification can be used:

$$(env = v) \mapsto term$$

$$term \Rightarrow LS$$

where *term* is a predicate describing that the system is stable until the environment changes. Moreover, *term* describes a set of states of the system such that each state in the set is legitimate and successor states of a state in the set are also in the set until the environment changes.

4.2 Specifications of Solution Strategies

After the problem specification is formulated, a solution strategy to solve the problem must be specified. Next, the solution strategy must be refined until it is restrictive enough to be directly translated into UNITY code. *Leads-to* properties can be refined to *ensures* properties in order to directly obtain the program text satisfying the *leads-to* properties. Similarly, *eventually-implies* properties can be refined to *assures* properties in order to directly obtain the program text satisfying the *eventually-implies* properties. This section describes how the solution strategies can be formulated for a special class of distributed self-stabilizing algorithms.

Define relation $\lhd$ on states by: $a \lhd b$ iff the state of node b is dependent upon the state of node a. The relation $\lhd^*$ is defined as the reflexive transitive closure of $\lhd$. Then a solution strategy has the *non-circularity* property iff $\lhd^*$ is antisymmetric, that is $\lhd^*$ is a partial order relation.

Let $env(b)$ denote the local environment of b, that is that part of the environment the state of node b depends upon. All local environments together define the (global) environment. Define $term(b)$ to be a predicate to denote that every node $a \lhd^* b$ has stabilized (and is in a legitimate state). Then a solution strategy having the non-circularity property, thus $\lhd^*$ is antisymmetric, can generally be specified as follows:

$$(\langle \forall a \ : \ a \lhd b \ :: \ term(a) \rangle \ \wedge \ (env(b) = v_b)) \mapsto term(b)$$

In the next section, after the problem specification is presented, a non-circular solution strategy is specified and proven correct, i.e. every algorithm satisfying this solution strategy solves the problem as specified in the problem specification. Next, the specification is refined to *assures* properties from which a program text directly follows.

5 Example

Assume that every node in a network has a data-item, called its *input*. Inputs are not necessarily unique elements, and are taken from a totally ordered set. The input of a node may change spontaneously. This section formally derives and verifies a distributed algorithm to maintain the maximum of all inputs in the network. In an earlier article [12], we formally derived a similar algorithm without using the new operators. It turns out that the use of the new operators clarifies the description considerably.

Section 5.1 formally presents the problem, using the (general) specification from Section 4.1. The environment corresponds to the bag of inputs and in a legitimate state it is guaranteed that a dedicated variable has as value the maximum of the inputs.

Next, Section 5.2 formally presents a solution strategy, using the general specification from Section 4.2, on a graph upon which a spanning tree is imposed. The local environment of a node corresponds to the input associated with the node. The state of every node in the spanning tree depends upon its local environment and the states of its sons. Furthermore, the solution strategy is proven correct, i.e. satisfying the problem specification.

Section 5.3 refines this strategy to a specification sufficiently restrictive for the program text to be directly obtainable from it.

5.1 Maximum Maintenance on a Graph: Problem Specification

This section formally specifies the problem. Before this specification is given, some notation is introduced.

5.1.1 Notation

Here is some notation and some variables (with their intended use).

$G = (N, E)$ denotes the finite graph corresponding to a network. Every node $a \in N$ corresponds to a processor in the network. Moreover, every node can be uniquely identified and will be denoted by small letters. Every edge $\{a, b\} \in E$ corresponds to a (bidirectional) link between processors a and b. These links are assumed to be fault-free and have a FIFO behaviour. Furthermore, it is assumed that the graph is *connected*.

For each node a, there is a variable $inp(a)$; $inp(a)$ contains the input of node a.

There is a variable $m(G)$; it is intended to hold the maximum of the inputs.

Expression $nodesval(G)$ is short for the bag of $(a, inp(a))$ where a ranges over the nodes; i.e.,

$$nodesval(G) = \langle \cup a \ : a \in N \ :: (a, inp(a)) \rangle.$$

$\vec{M}$ denotes an arbitrary value of the type of $nodesval(G)$; we use it as a free variable in a specification.

There is a predicate *term*; $term(G)$ denotes that the system has *terminated*. For the time being we do not define this predicate in full, but only partially define it by requiring it to satisfy the following requirement:

$$term(G) \ \Rightarrow \ m(G) = \langle \mathbf{max}\, a \ : \ a \in N \ :: \ inp(a) \rangle.$$

5.1.2 Specification

An obvious way of specifying the problem (in temporal logic) of maintaining the maximum value of the inputs in a network G, where each node has an input, is:

$$(\Diamond\Box(nodesval(G) = \vec{M})) \Rightarrow (\Diamond\Box(m(G) = \langle \mathbf{max}\, a \ : \ a \in N \ :: \ inp(a) \rangle)).$$

Section 4 gives a general UNITY specification to specify a problem which is formally presented with the above displayed temporal logic specification. This results in the following specification:

SP1: $(nodesval(G) = \vec{M}) \mapsto term(G)$

In other words, given an indexed set of inputs, eventually the algorithm terminates (and thus "delivers" the maximum input) or the indexed set of inputs changes. And once the algorithm terminates, this remains to hold until at least one input changes.

5.2 Solution Strategy: Maximum Maintenance on a Graph using a Spanning Tree

Before the solution strategy to solve the problem is presented, some more notation and assumptions are introduced.

5.2.1 Notation and Assumptions

Upon the graph $G = (N, E)$, a *spanning tree* is imposed. $R(G)$ denotes the root of the spanning tree. We assume that $a \lhd b$ holds if and only if (in the spanning tree) a is a son of b, and thus b is a father of a. If $a \lhd^* b$, we call a a *descendant of* b, and b an *ancestor of* a. Note that $\lhd^*$ is antisymmetric.

The inputs of descendants of node a are represented by $descval(a)$; i.e.,

$$descval(a) = \langle \cup b \ : b \lhd^* a \ :: \ (b, inp(b)) \rangle$$

Variable $m^a(a)$, which is called the *value of* a, is a's current view of the maximum of the inputs of the descendants of a. Moreover, $m^b(a)$ is b's view of $m^a(a)$ for every $a \lhd b$. (Think of $m^b(c)$ as located in node b, containing information about node c.)

The predicate $fatherknows(a)$ is defined by:

$$fatherknows(a) = ((a \lhd b) \Rightarrow (m^b(a) = m^a(a)))$$

Informally, $fatherknows(a)$ expresses that the value of a is known to its father.

The algorithm *has finished* for node a if and only if the variable $m^a(a)$ is the maximum of its input and all the values of its sons, and the algorithm has terminated (which is defined next) for these sons.

$$\begin{aligned} &hfinished(a) = \\ &\quad m^a(a) = \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b \ : \ b \lhd a \ :: \ m^a(b) \rangle\} \ \wedge \\ &\quad \langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle \end{aligned}$$

The algorithm *has terminated* for node a if and only if the algorithm has finished for a and this value of a is known to the father of a.

$$hterm(a) = (hfinished(a) \wedge fatherknows(a)).$$

Because of the connectedness of G, there exists a spanning tree on G. The definitions imply the following:

$$(a \lhd^* R(G)) = (a \in N)$$

$$nodesval(G) = descval(R(G))$$

$$hfinished(R(G)) = hterm(R(G))$$

5.2.2 Specification

Consider the following derivation (the top line being the ultimate goal):

$$\begin{aligned} & m(G) = \langle \mathbf{max}\, a \ : \ a \in N \ :: \ inp(a) \rangle \\ = \quad & \{\text{Assume } m^{R(G)}(R(G)) = m(G), \text{ and notice } (a \in N) = (a \lhd^* R(G))\} \\ & m^{R(G)}(R(G)) = \langle \mathbf{max}\, a \ : \ a \lhd^* R(G) \ :: \ inp(a) \rangle \\ \Leftarrow \quad & \{R(G) \in N\} \\ & \langle \forall a \ : \ a \in N \ :: \ m^a(a) = \langle \mathbf{max}\, b \ : \ b \lhd^* a \ :: \ inp(b) \rangle \rangle \qquad (1) \\ = \quad & \{\text{Unfolding using definition } \lhd^*\} \\ & \langle \forall a \ : \ a \in N \ :: \ m^a(a) = \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b \ : \ b \lhd^+ a \ :: \ inp(b) \rangle\} \rangle \end{aligned}$$

$=$ {Unfolding using definition $\lhd^+$}

$\langle \forall a \ : \ a \in N \ :: \ m^a(a) = \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b \ : \ b \lhd a \ :: \ \langle \mathbf{max}\, c \ : \ c \lhd^* b \ :: \ inp(c)\rangle\rangle\}\rangle$

$=$ {Folding using 1}

$\langle \forall a \ : \ a \in N \ :: \ m^a(a) = \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b \ : \ b \lhd a \ :: m^b(b)\rangle\}\rangle$

The above derivation shows that that to maintain the maximum of all inputs in a graph (upon which a spanning tree is imposed), it is sufficient that each node in the graph maintains the maximum of all inputs at its descendants. The root of the graph then maintains the maximum of all inputs in the graph. In turn, this can be accomplished by maintaining at each node the maximum of its input and the values at its sons.

Thus, if each son of a node has terminated (such that each son has a value which is the maximum of the inputs at its descendants and known to the node) and the node has computed the maximum of its input and the (copies of the) values of its sons, the node has computed the maximum of the inputs at its descendants (in which case the node has finished its computation).

Moreover, for a node to know the maximum of the inputs at the descendants of a son, it is sufficient that after the son has finished its computation, in which case it is aware of the maximum of the inputs at its descendants, the value of the son is known to the node.

These observations suggest the following specification, consisting of two rules, SP2 and SP3.

SP2: $(\langle \forall b \ : \ b \lhd a \ :: \ hterm(b)\rangle \wedge (inp(a) = k)) \mapsto hfinished(a)$

In words, this rule formalises the following two properties.

1. After every descendant of each son of a has successfully computed its value (i.e. has computed the maximum of the inputs at its descendants) and a is aware of the value of each son ($\langle \forall b \ : \ b \lhd a \ :: \ hterm(b)\rangle$ holds), then within finite time node a computes the maximum of the inputs at its descendants such that $hfinished(a)$ holds, the value of the input at a changes or at least one descendant of a son of a must recompute its value ($\neg\langle \forall b \ : \ b \lhd a \ :: \ hterm(b)\rangle$ holds).

2. And when every descendant of node a has successfully computed its value ($hfinished(a)$ holds), then this continues to hold until the value of the input at a changes or at least one descendant of a son of a must recompute its value ($\neg\langle \forall b \ : \ b \lhd a \ :: \ hterm(b)\rangle$ holds).

SP3: $hfinished(a) \mapsto hterm(a)$

In words, this rule formalises the following two properties.

1. When every descendant of node a has successfully computed its value ($hfinished(a)$ holds), then within finite time the father of a is notified of the maximum value of the inputs at the descendants of a such that $hterm(a)$ holds or at least one descendant of a must recompute its value ($\neg hfinished(a)$ holds).

2. And when every descendant of a has successfully computed its value and the father of a is aware of the value of a ($hterm(a)$ holds), then this continues to hold until at least one descendant of a must recompute its value ($\neg hfinished(a)$ holds).

5.2.3 Proof of Correctness

We prove that the problem specification (SP1) is met by any solution strategy satisfying SP2 and SP3, that is, SP1 $\Leftarrow$ SP2 $\wedge$ SP3.

Lemma 5.1 $hfinished(a) = [(m^a(a) = \langle \mathbf{max}\, c \ : \ c \lhd^* a \ :: \ inp(c)\rangle) \ \wedge \ \langle \forall b \ : \ b \lhd a \ :: \ hterm(b)\rangle]$.

Proof: We use induction on the structure of the spanning tree.
Base: a is a leaf of the spanning tree. Then:

$$
\begin{array}{ll}
& \mathit{hfinished}(a) \\
= & \quad \{\text{Definition of } \mathit{hfinished}\} \\
& ((m^a(a) = \mathbf{max}\,\{\mathit{inp}(a), \langle \mathbf{max}\, b \;:\; b \lhd a \;::\; \mathit{inp}(b)\rangle\}) \;\wedge\; \langle \forall b \;:\; b \lhd a \;::\; \mathit{hterm}(b)\rangle) \\
= & \quad \{a \text{ is a leaf, so } \neg\langle \exists b \;:\; b \in N \;::\; b \lhd a\rangle\} \\
& ((m^a(a) = \mathit{inp}(a)) \;\wedge\; \langle \forall b \;:\; b \lhd a \;::\; \mathit{hterm}(b)\rangle) \\
= & \quad \{a \text{ is a leaf}\} \\
& ((m^a(a) = \langle \mathbf{max}\, c \;:\; c \lhd^* a \;::\; \mathit{inp}(c)\rangle) \;\wedge\; \langle \forall b \;:\; b \lhd a \;::\; \mathit{hterm}(b)\rangle)
\end{array}
$$

Step: Suppose a is not a leaf and the lemma holds for all sons of a, i.e.

$$
\begin{array}{l}
\langle \forall b \;:\; b \lhd a \;:: \\
\qquad \mathit{hfinished}(b) = ((m^b(b) = \langle \mathbf{max}\, c \;:\; c \lhd^* b \;::\; \mathit{inp}(c)\rangle) \;\wedge\; \langle \forall d \;:\; d \lhd b \;::\; \mathit{hterm}(d)\rangle) \\
\rangle
\end{array}
$$

Then the following holds:

$$
\begin{array}{ll}
& \mathit{hfinished}(a) \\
= & \quad \{\text{Definition of } \mathit{hfinished}\} \\
& (m^a(a) = \mathbf{max}\,\{\mathit{inp}(a), \langle \mathbf{max}\, b \;:\; b \lhd a \;::\; m^a(b)\rangle\}) \;\wedge\; \langle \forall b \;:\; b \lhd a \;::\; \mathit{hterm}(b)\rangle \\
= & \quad \{\mathit{hterm}(b) \Rightarrow (m^a(b) = m^b(b))\} \\
& (m^a(a) = \mathbf{max}\,\{\mathit{inp}(a), \langle \mathbf{max}\, b \;:\; b \lhd a \;::\; m^b(b)\rangle\}) \;\wedge\; \langle \forall b \;:\; b \lhd a \;::\; \mathit{hterm}(b)\rangle \\
= & \quad \{\mathit{hterm}(b) \Rightarrow \mathit{hfinished}(b), \text{ Induction Hypothesis}\} \\
& (m^a(a) = \mathbf{max}\,\{\mathit{inp}(a), \langle \mathbf{max}\, b \;:\; b \lhd a \;::\; \langle \mathbf{max}\, c \;:\; c \lhd^* b \;::\; \mathit{inp}(c)\rangle\rangle\}) \;\wedge \\
& \langle \forall b \;:\; b \lhd a \;::\; \mathit{hterm}(b)\rangle \\
= & \quad \{\text{Rewriting}\} \\
& (m^a(a) = \langle \mathbf{max}\, c \;:\; c \lhd^* a \;::\; \mathit{inp}(c)\rangle) \;\wedge\; \langle \forall b \;:\; b \lhd a \;::\; \mathit{hterm}(b)\rangle
\end{array}
$$

This proves the induction step, thereby proving the lemma.
□

In order to find a formal definition for $\mathit{term}(G)$, such that the requirement of Section 5.1.1 is met, we argue as follows.

$$
\begin{array}{ll}
& \text{“Requirement of Section 5.1.1”} \\
= & \\
& \mathit{term}(G) \Rightarrow (m(G) = \langle \mathbf{max}\, a \;:\; a \in N \;::\; \mathit{inp}(a)\rangle) \\
= & \quad \{\text{Section 5.2.1: } (a \in N) = (a \lhd^* R(G))\} \\
& \mathit{term}(G) \Rightarrow (m(G) = \langle \mathbf{max}\, a \;:\; a \lhd^* R(G) \;::\; \mathit{inp}(a)\rangle) \\
\Leftarrow & \quad \{\text{Lemma 5.1, with } a = R(G)\} \\
& \mathit{term}(G) \Rightarrow ((m(G) = m^{R(G)}(R(G))) \wedge \mathit{hfinished}(R(G))) \\
= & \quad \{\text{Define } m(G) = m^{R(G)}(R(G))\} \\
& \mathit{term}(G) \Rightarrow \mathit{hfinished}(R(G))
\end{array}
$$

Hence, defining
$m(G) = m^{R(G)}(R(G))$
$\mathit{term}(G) = \mathit{hfinished}(R(G))$
$\qquad = \mathit{hterm}(R(G))$
the requirement is fulfilled.

Theorem 5.1 *SP1 is met by SP2 and SP3.*

Proof:

SP1

$=$ {Definition of SP1}

$(nodesval(G) = \vec{M}) \mapsto term(G)$

$=$ {Definition of *nodesval* and *term*}

$(descval(R(G)) = \vec{M}) \mapsto hfinished(R(G))$

$\Leftarrow$ {Generalization}

$\langle \forall a \ : \ a \in N \ :: (descval(a) = \vec{M}_a) \mapsto hfinished(a) \rangle$

$=$ {Proved below, using SP2 and SP3}

true

The last step is shown by induction on the structure of the spanning tree.

Let $\vec{M}_a = (\{(a, inp(a))\} \cup \langle \cup b \ : \ b \lhd a \ :: \ descval(b) \rangle)$

Base: a is a leaf.

SP2

$=$ {Definition of SP2}

$(\langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle \wedge (inp(a) = k)) \mapsto hfinished(a)$

$=$ {a is a leaf of the spanning-tree}

$(inp(a) = k) \mapsto hfinished(a)$

$=$ {Rewriting the left side}

$(descval(a) = \vec{M}_a) \mapsto hfinished(a)$

Step: Suppose a is not a leaf and

$$\langle \forall b \ : \ b \lhd a \ :: (descval(b) = \vec{M}_b) \mapsto hfinished(b) \rangle$$

Then the following holds:

$\langle \forall b \ : \ b \lhd a \ :: \ (descval(b) = \vec{M}_b) \mapsto hfinished(b) \rangle$

$\Rightarrow$ {Generalized Conjunction theorem}

$\langle \forall b \ : \ b \lhd a \ :: \ (descval(b) = \vec{M}_b) \rangle \mapsto \langle \forall b \ : \ b \lhd a \ :: \ hfinished(b) \rangle$

$\Rightarrow$ {Precondition Strengthening}

$(\langle \forall b \ : \ b \lhd a \ :: \ descval(b) = \vec{M}_b \rangle \ \wedge \ (inp(a) = k)) \mapsto \langle \forall b \ : \ b \lhd a \ :: \ hfinished(b) \rangle$

$=$ {Rewriting the left side}

$(descval(a) = \vec{M}_a) \mapsto \langle \forall b \ : \ b \lhd a \ :: \ hfinished(b) \rangle$ **(2)**

We continue by rewriting SP3.

SP3

$=$ {Definition of SP3}

$\langle \forall b \ : \ b \lhd a \ :: \ hfinished(b) \mapsto hterm(b) \rangle$

$\Rightarrow$ {Generalized Conjunction theorem}

$\langle \forall b \ : \ b \lhd a \ :: \ hfinished(b) \rangle \mapsto \langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle$ **(3)**

Combining the above derivations results in the following.

$$
\begin{array}{ll}
 & (\langle \forall b \ : \ b \lhd a \ :: \ (descval(b) = \vec{M}_b) \mapsto hfinished(b) \rangle) \wedge \text{SP3} \\
\Rightarrow & \quad \{2 \text{ and } 3\} \\
 & ((descval(a) = \vec{M}_a) \mapsto \langle \forall b \ : \ b \lhd a \ :: \ hfinished(b) \rangle) \wedge \\
 & (\langle \forall b \ : \ b \lhd a \ :: \ hfinished(b) \rangle \mapsto \langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle) \\
\Rightarrow & \quad \{\text{Weak Transitivity theorem}\} \\
 & (descval(a) = \vec{M}_a) \mapsto (\langle \forall b \ : \ b \lhd a \ :: \ hfinished(b) \rangle \wedge \langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle) \\
= & \quad \{\text{Rewriting the right side}\} \\
 & (descval(a) = \vec{M}_a) \mapsto \langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle \\
= & \quad \{\text{Reflexivity}\} \\
 & ((descval(a) = \vec{M}_a) \mapsto \langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle) \wedge ((inp(a) = k) \mapsto (inp(a) = k)) \\
\Rightarrow & \quad \{\text{Conjunction theorem}\} \\
 & ((descval(a) = \vec{M}_a) \wedge (inp(a) = k)) \mapsto (\langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle \wedge (inp(a) = k)) \qquad (4)
\end{array}
$$

We conclude the induction step as follows.

$$
\begin{array}{ll}
 & \text{SP2} \wedge \text{SP3} \wedge (\langle \forall b \ : \ b \lhd a \ :: \ (descval(b) = \vec{M}_b) \mapsto hfinished(b) \rangle) \\
\Rightarrow & \quad \{4\} \\
 & \text{SP2} \wedge (((descval(a) = \vec{M}_a) \wedge (inp(a) = k)) \mapsto (\langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle \wedge (inp(a) = k))) \\
= & \quad \{\text{Definition of SP2}\} \\
 & ((\langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle \wedge (inp(a) = k)) \mapsto hfinished(a)) \wedge \\
 & (((descval(a) = \vec{M}_a) \wedge (inp(a) = k)) \mapsto (\langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle \wedge (inp(a) = k))) \\
\Rightarrow & \quad \{\text{Weak Transitivity theorem}\} \\
 & ((descval(a) = \vec{M}_a) \wedge (inp(a) = k)) \mapsto \\
 & \quad (hfinished(a) \wedge \langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle \wedge (inp(a) = k)) \\
\Rightarrow & \quad \{\text{Restricted Consequence Weakening}\} \\
 & ((descval(a) = \vec{M}_a) \wedge (inp(a) = k)) \mapsto (hfinished(a) \wedge \langle \forall b \ : \ b \lhd a \ :: \ hterm(b) \rangle) \\
= & \quad \{\text{Rewriting the right side}\} \\
 & ((descval(a) = \vec{M}_a) \wedge (inp(a) = k)) \mapsto hfinished(a) \\
= & \quad \{\text{Rewriting the left side}\} \\
 & (descval(a) = \vec{M}_a) \mapsto hfinished(a)
\end{array}
$$

This proves the induction step, thereby proving the theorem.
□

5.3 Refinement Step: Towards the Implementation

Opposed to the *ensures* (and therefore also to *assures* relations), a *leads-to* and *eventually-implies* relation cannot be translated directly into a program statement. Thus, the solution strategy given above is not yet in a form that can be directly translated into a program text. In order to obtain such a specification, conditions SP2 and SP3 must be refined. Furthermore, to be able to implement the resulting program on a distributed system, a statement within a process may only depend on its local state, i.e. information present within the process. For example, the execution of

a statement may not depend on the state of other nodes, such as the condition that the algorithm has terminated in some node. In this section a specification will be given that satisfies these constraints.

5.3.1 Specification

As already stated in Section 5.2.2, specification rule SP2 formalises the following two properties.

1. After every descendant of each son of a has successfully computed its value and a is aware of the value of each son, then within finite time node a has successfully computed the maximum of the inputs at its descendants, or the value of the input at a has changed or at least one descendant of a son of a must recompute its value.

2. And when every descendant of node a has successfully computed its value, then this continues to hold until the value of the input at a changes or at least one descendant of a son of a must recompute its value.

The first property can be satisfied by ensuring that at any time each node eventually computes the maximum of its input and the values at its sons. Then, after every descendant of each son of node a has successfully computed its value, it is assured that eventually a will recompute its value. After this recomputation, node a has successfully computed its value or at least one descendant of a son of a has an incorrect value and thus must recompute its value.

The second property can be satisfied by assuring that once a node has computed the maximum of its input and the values of its sons, this must continue to hold until its input or a value of a son has changed.

These observations suggest the following rule:

SP4: true $assures_p\ (m^a(a) = \max\{inp(a), \langle \max b\ :\ b \lhd a\ ::\ m^a(b)\rangle\})$
where
$p = ((inp(a) = k) \wedge \langle \forall b\ :\ b \lhd a\ ::\ m^a(b) = m_b\rangle)$

Specification rule SP3, as given in Section 5.2.2, formalises the following two properties.

1. When every descendant of node a has successfully computed its value, then within finite time the father of a is notified of the maximum value of the inputs at the descendants of a or at least one descendant of a must recompute its value.

2. And when every descendant of a has successfully computed its value and the father of a is aware of the value of a, then this continues to hold until at least one descendant of a must recompute its value.

The first property can be satisfied by ensuring that at any time the father of each node a is eventually notified of the value of a. Then, it is guaranteed that if a has successfully computed its value, within finite time the father of a is notified of the value of a. After the notification, the father of a is aware of the maximum value of the inputs at the descendants of a or at least one descendant of a must recompute its value.

The second property can be satisfied by assuring that once the father of node a is aware of the value of a, this father may not change its view of the value of a until the value of a changes.

These observations suggest the following relation:

SP5: true $assures_{(m^a(a)=m_a)}\ fatherknows(a)$

5.3.2 Proof of Correctness

To prove the correctness of the refinement, we prove that the solution strategy, given by SP2 and SP3 is satisfied by any strategy satisfying SP4 and SP5.

Theorem 5.2 *SP2 is met by SP4 and SP5.*

Proof:

SP4

$=$ {Definition of SP4}

true $assures_p$ $(m^a(a) = \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b : b \lhd a :: m^a(b)\rangle\})$

where

$p = ((inp(a) = k) \wedge \langle \forall b : b \lhd a :: m^a(b) = m_b\rangle)$

$\Rightarrow$ {*assures* promotion}

$((inp(a) = k) \wedge \langle \forall b : b \lhd b :: m^a(b) = m_b\rangle) \mapsto$
$(m^a(a) = \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b : b \lhd a :: m^a(b)\rangle\})$

$=$ {Reflexivity}

$(((inp(a) = k) \wedge \langle \forall b : b \lhd a :: m^a(b) = m_b\rangle) \mapsto$
$(m^a(a) = \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b : b \lhd a :: m^a(b)\rangle\})) \wedge$
$(\langle \forall b : b \lhd a :: hterm(b)\rangle \mapsto \langle \forall b : b \lhd a :: hterm(b)\rangle)$

$\Rightarrow$ {Conjunction theorem}

$((inp(a) = k) \wedge \langle \forall b : b \lhd a :: m^a(b) = m_b\rangle \wedge \langle \forall b : b \lhd a :: hterm(b)\rangle) \mapsto$
$((m^a(a) = \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b : b \lhd a :: m^a(b)\rangle\}) \wedge \langle \forall b : b \lhd a :: hterm(b)\rangle)$

$=$ {Rewriting the right side}

$((inp(a) = k) \wedge \langle \forall b : b \lhd a :: m^a(b) = m_b\rangle \wedge \langle \forall b : b \lhd a :: hterm(b)\rangle) \mapsto hfinished(a)$

$=$ {$(b \lhd a \wedge hterm(b)) \Rightarrow (hfinished(b) \wedge m^a(b) = m^b(b))$, Lemma 5.1}

$(\langle \forall b : b \lhd a :: hterm(b)\rangle \wedge (inp(a) = k)) \mapsto hfinished(a)$

□

Theorem 5.3 *SP3 is met by SP4 and SP5.*

Proof:

SP5

$=$ {Definition of SP5}

true $assures_{(m^a(a)=m_a)}$ $fatherknows(a)$

$\Rightarrow$ {*assures* promotion}

$(m^a(a) = m_a) \mapsto fatherknows(a)$

$=$ {Reflexivity}

$((m^a(a) = m_a) \mapsto fatherknows(a)) \wedge (hfinished(a) \mapsto hfinished(a))$

$\Rightarrow$ {Conjunction theorem}

$((m^a(a) = m_a) \wedge hfinished(a)) \mapsto (hfinished(a) \wedge fatherknows(a))$

$=$ {Rewriting the right side}

$((m^a(a) = m_a) \wedge hfinished(a)) \mapsto hterm(a)$

$=$ {Lemma 5.1}

$hfinished(a) \mapsto hterm(a)$

□

5.3.3 Derivation of a Program from the Specification

A program follows directly from the specification of the solution strategy. The function $inp(a)$ delivers the input of some node a.

SP4 suggests the recomputation of $m^a(a)$:

$$\langle \mathbf{I}a \ :: \ m^a(a) := \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b \ : \ b \lhd a \ :: \ m^a(b)\rangle\}\rangle$$

SP5 suggests that every node is aware of the value of each of its sons:

$$\langle \mathbf{I}a, f \ : \ a \lhd f \ :: \ m^f(a) := m^a(a)\rangle.$$

To show that these statements do not violate the safety properties (unless-parts of SP4 and SP5) is rather straightforward and left to the reader.

The complete program follows.

Program {maximum computation on a spanning tree}
assign
$\langle \mathbf{I}a, f \ : \ a \lhd f \ ::$
$m^f(a) := m^a(a)$
I
$m^a(a) := \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b \ : \ b \lhd a \ :: \ m^a(b)\rangle\}$
$\rangle$
end {maximum computation on a spanning tree}

5.3.4 Remarks

The program given in Section 5.3.3 can be implemented on a distributed system in an obvious way. The variables $m^a(b)$ and $m^a(a)$ are local to a. The assignment $m^f(a) := m^a(a)$ is done by communicating the value of $m^a(a)$ over the channel between a and f. How channels are represented in UNITY, has been described in [4, 12].

Alternatively, SP4 could be replaced by the following rules:

SP4a: (a is a leaf) $assures_{inp(a)=k)}\ (m^a(a) = inp(a))$

SP4b: (a is not a leaf) $assures_p\ (m^a(a) = \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b \ : \ b \lhd a \ :: \ m^a(b)\rangle\})$
where
$p = ((inp(a) = k) \wedge \langle \forall b \ : \ b \lhd a \ :: \ m^a(b) = m_b\rangle)$

It can easily be proven that SP4a and SP4b are equivalent to SP4, i.e. every algorithm that satisfies SP4a and SP4b also satisfies SP4, and vice versa. From SP4a and SP4b, the following assignments result:

$$\langle \mathbf{I}a \ : \ a \text{ a leaf} \ :: \ m^a(a) := inp(a)\rangle$$

and

$$\langle \mathbf{I}a \ : \ a \text{ not a leaf} \ :: \ m^a(a) := \mathbf{max}\,\{inp(a), \langle \mathbf{max}\, b \ : \ b \lhd a \ :: \ m^a(b)\rangle\}\rangle$$

6 Conclusion

In [12] a first attempt was made to formally design and verify a self-stabilizing algorithm in UNITY. In the specification an ever-recurring pattern occurred. In this article, this pattern is captured in one single operator: $\mapsto$ (eventually-implies). This operator is defined in terms of the basic UNITY operators. It consists of two parts: a progress (*leads-to*) and a safety (*unless*) part. The progress part can be used to describe that always within finite time a legitimate state is reached or the environment is changed. The safety part can be used to describe that once a legitimate state has been reached, the following states remain legitimate until the environment changes. Theorems have been given that have proven to be helpful in calculating with this new operator.

A general scheme has been presented to specify the problem a self-stabilizing algorithm is assumed to solve. Also a general scheme has been given to specify a solution strategy of a restricted class of self-stabilizing algorithms. These algorithms do satisfy the non-circularity property.

An example has been given of the formal derivation and verification of a self-stabilizing algorithm, with the non-circularity property, to maintain the maximum value of a bag of inputs. A specification of a solution strategy was proven to be correct, that is each program satisfying the specification of the solution strategy also satisfies the problem specification. Next, the specification of the solution strategy was refined to a specification that directly suggests a program text.

The final specification was given in terms of another newly introduced operator: *assures*. Operator *assures* describes the behaviour of a process which is part of the resulting self-stabilizing algorithm. As is the case with *ensures*, from the *assures* properties, program text can be directly obtained. Also *assures* consists of a progress (*ensures*) and a safety (*unless*) part. The progress part describes that the process will always reach a, for that process, legitimate state by executing one statement, whatever the values of the environment and states of other processes are. The safety part describes that once the process has reached a legitimate state, the state remains legitimate until the environment or states of other processes (the state of the process is dependent upon) changes.

The problem of maintaining the maximum value of a bag of inputs as such does not seem to be of great interest. However, it is our hope that the techniques used in the formal design and verification of this algorithm will be helpful in the formal design of more self-stabilizing algorithms, especially routing algorithms. The problem of maintaining the maximum value of a (possibly changing) bag of inputs and the problem of keeping topological information about a (dynamical) network bear some similarities. The network can be considered as a set of nodes. With each node in this set corresponds a (possibly changing) set of paths from a source node to that node. Every node must, for every other destination node in the set, maintain the optimal (i.e. minimum length) path.

At first sight, routing algorithms do not satisfy the non-circularity property. For example, in the routing algorithm as proposed by Tajibnapis [27] every node maintains a copy of the routing table of each neighbouring node. For each node, these copies are used to calculate its routing table. In other words, the data maintained by a node depends upon the data at its neighbours, and vice versa. However, by studying the algorithm more carefully, the following non-circularity property can be discovered. The algorithm computes successively the shortest paths of increasing length. Consequently, the *correct* routing information for some destination node d at node a does not depend upon the routing information for d at a neighbour node b of a if the distance from a to d is at most the distance from b to d.

7 Acknowledgement

The authors thank Arjen Uittenbogaard, Kaisa Sere, Rob Udink, Frans Rietman, Nico Verwer and the members of the Calculi for Distributed Program Construction Club for their help during the course of this research.

References

[1] A. Arora and M.G. Gouda. Distributed reset (extended abstract). In *Proceedings of 10th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Springer-Verlag, 1990.

[2] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *Proceedings of 32'nd IEEE Symposium on Foundations on Computer Science*, October 1991.

[3] J. Burns, M. Gouda, and R. Miller. Stabilization and pseudo-stabilization. Technical Report TR-90-13, University of Texas, May 1990.

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

[5] H.C. Cunningham and G. Roman. A unity-style programming logic for a shared dataspace language. Technical Report WUCS-89-5, Department of computer science Washington University, March 1989.

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

[7] J.M. Jaffe, A.E. Baratz, and A. Segall. Subtle design issues in the implementation of distributed, dynamic routing algorithms. *Computer Networks and ISDN Systems*, (12):147–158, 1986.

[8] S. Katz and K.J. Perry. Self-stabilizing extensions for message-passing systems. In *Proceedings of the Ninth Annual ACM Symposium on Principles of Distributed Computing*, pages 91–101, Quebec City, August 1990.

[9] E. Knapp. An exercise in the formal derivation of parallel programs: Maximum flows in graphs. *ACM Transactions on Programming Languages and Systems*, 12(2):203–223, April 1990.

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

[11] L. Lamport. An assertional correctness proof of a distributed algorithm. *Science of Computer Programming*, (2):175–206, 1982.

[12] P.J.A. Lentfert and S.D. Swierstra. Distributed maximum maintenance on hierarchically divided graphs. *Formal Aspects of Computing*, 1992.

[13] P.J.A. Lentfert, A.H. Uittenbogaard, S.D. Swierstra, and G. Tel. Distributed hierarchical routing. Technical Report RUU-CS-89-5, Utrecht University, March 1989. Also in: P.M.G. Apers et al. (Eds.). *Proceedings CSN89.* Utrecht, November 9–10, 1989.

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 1. Springer Verlag, to appear.

[15] J.M. McQuillan, I. Richer, and E.C. Rosen. The new routing algorithm for the arpanet. *IEEE Transactions on Communications*, com-28(5):711–719, May 1980.

[16] J. Misra. Specifications of concurrently accessed data. In J. L. A. van de Snepscheut, editor, *Proceedings of the Conference on the Mathematics of Program Construction, Groningen*, pages 91–114, New York, June 1989. Springer-Verlag.

[17] J. Pachl. Three definitions of *leads-to* for unity. *Notes on UNITY*, (23-90), 1990.

[18] J. Pachl. A simple proof of a completeness result for *leads-to* in the unity logic. *Information Processing Letters*, (41):35–38, 1992.

[19] R. Perlman. Fault-tolerant broadcast of routing information. *Computer Networks*, 7:395–405, December 1983.

[20] J. Rao. On a notion of completeness for the leads-to. *Notes on UNITY*, (24-90), 1991.

[21] G. Roman and H.C. Cunningham. The synchronic group: A concurrent programming concept and its proof logic. Technical Report WUCS-89-43, Department of computer science, Washington University, October 1989.

[22] B.A. Sanders. Stepwise refinement of mixed specifications of concurrent programs. In M. Broy and C.B. Jones, editors, *Proc. IFIP Working Conf. on Programming and Methods*, pages 1–25. Elsevier Science Publishers B.V. (North Holland), May 1990.

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

[24] A. Segall. Distributed network protocols. *IEEE Trans. Inf. Theory*, IT-29(1):23–35, January 1983.

[25] A.K. Sing. Specification of concurrent objects using auxiliary variables. *Science of Computer Programming*, (16):49–88, 1991.

[26] M. G. Staskauskas. The formal specification and design of a distributed electronic funds transfer system. *IEEE Trans. Comp.*, 37(12):1515–1528, December 1988.

[27] W.D. Tajibnapis. A correctness proof of a topology information maintenance protocol for a distributed computer network. *Computer Systems*, 20(7):477–485, July 1977.

A Theorems about UNITY Operators

In this section the theorems about *assures* and $\mapsto\!\!\!\!\!\cdot$, as stated in Section 3, are proven to be correct.

Theorem A.1 *assures promotion:*

$$\frac{r \textit{ assures}_p \ q}{(p \wedge r) \mapsto\!\!\!\!\!\cdot \ q}$$

Proof:

$r \textit{ assures}_p \ q$

$\Rightarrow$ {Definition of *assures*}

$r \textit{ ensures } q$

$\Rightarrow$ {*ensures* promotion}

$r \mapsto q$

$=$ {Predicate logic}

$(r \mapsto q) \wedge ((p \wedge r) \Rightarrow r)$

$\Rightarrow$ {Implication theorem}

$(r \mapsto q) \wedge ((p \wedge r) \mapsto r)$

$\Rightarrow$ {Transitivity theorem}

$(p \wedge r) \mapsto q$

$\Rightarrow$ {Consequence Weakening}

$(p \wedge r) \mapsto q \vee \neg(p \wedge r)$ (5)

$r \textit{ assures}_p \ q$

$\Rightarrow$ {Definition of *assures*}

$(p \wedge q \wedge r) \textit{ unless } \neg p$

$\Rightarrow$ {Consequence Weakening}

$(p \wedge q \wedge r) \textit{ unless } \neg(p \wedge r)$ (6)

The conjunction of 5 and 6 is exactly the definition of $(p \wedge r) \mapsto\!\!\!\!\!\cdot \ q$, which concludes the theorem. □

Theorem A.2 *Implication:*

$$\frac{p \Rightarrow q}{p \mapsto\!\!\!\!\!\!\cdot\; q}$$

Proof:

$p \Rightarrow q$
$\Rightarrow$ {Implication theorem}
$p \mapsto q$
$\Rightarrow$ {Consequence Weakening}
$p \mapsto (q \vee \neg p)$ (7)

$p \Rightarrow q$
$=$ {Predicate logic}
$\neg(p \wedge q) \Rightarrow \neg p$
$=$ {Antireflexivity of *unless*}
$(\neg(p \wedge q) \Rightarrow \neg p) \wedge ((p \wedge q)$ *unless* $\neg(p \wedge q))$
$\Rightarrow$ {Consequence Weakening}
$(p \wedge q)$ *unless* $\neg p$ (8)

The conjunction of 7 and 8 is exactly the definition of $p \mapsto\!\!\!\!\!\!\cdot\; q$, which concludes the theorem.
□

Theorem A.3 *Reflexivity:*

$$p \mapsto\!\!\!\!\!\!\cdot\; p$$

Proof: Follows directly from the Implication theorem.
□

Theorem A.4 *Weak Transitivity:*

$$\frac{(p \mapsto\!\!\!\!\!\!\cdot\; q)\ ,\ (q \mapsto\!\!\!\!\!\!\cdot\; r)}{p \mapsto\!\!\!\!\!\!\cdot\; (q \wedge r)}$$

Proof:

$(p \mapsto\!\!\!\!\!\!\cdot\; q) \wedge (q \mapsto\!\!\!\!\!\!\cdot\; r)$
$\Rightarrow$ {Definition of $\mapsto\!\!\!\!\!\!\cdot$}
$(p \mapsto (q \vee \neg p)) \wedge (q \mapsto (r \vee \neg q))$
$\Rightarrow$ {Cancellation theorem}
$p \mapsto (r \vee \neg q \vee \neg p)$ (9)

$(p \mapsto\!\!\!\!\!\!\cdot\; q)$
$\Rightarrow$ {Definition of $\mapsto\!\!\!\!\!\!\cdot$}
$(p \wedge q)$ *unless* $\neg p$ (10)

$(p \mapsto\!\!\!\!\!\!\cdot\; q) \wedge (q \mapsto\!\!\!\!\!\!\cdot\; r)$
$\Rightarrow$ {9 and 10}
$(p \mapsto (r \vee \neg q \vee \neg p)) \wedge ((p \wedge q)$ *unless* $\neg p)$
$\Rightarrow$ {PSP theorem}
$(p \wedge q) \mapsto (((r \vee \neg q \vee \neg p) \wedge (p \wedge q)) \vee \neg p)$
$=$ {Rewriting the right side}
$(p \wedge q) \mapsto ((r \wedge q) \vee \neg p)$ (11)

$p \mapsto\!\!\!\!\!\!\cdot\;\; q$

$\Rightarrow$ {Definition of $\mapsto\!\!\!\!\!\!\cdot$}

$p \mapsto ((p \wedge q) \vee \neg p)$ (12)

$(p \mapsto\!\!\!\!\!\!\cdot\;\; q) \wedge (q \mapsto\!\!\!\!\!\!\cdot\;\; r)$

$\Rightarrow$ {11 and 12}

$(p \mapsto ((p \wedge q) \vee \neg p)) \wedge ((p \wedge q) \mapsto ((r \wedge q) \vee \neg p))$

$\Rightarrow$ {Cancellation theorem}

$p \mapsto ((q \wedge r) \vee \neg p)$ (13)

$(p \mapsto\!\!\!\!\!\!\cdot\;\; q) \wedge (q \mapsto\!\!\!\!\!\!\cdot\;\; r)$

$\Rightarrow$ {Definition of $\mapsto\!\!\!\!\!\!\cdot$}

$((p \wedge q)$ *unless* $\neg p) \wedge ((q \wedge r)$ *unless* $\neg q)$

$\Rightarrow$ {Conjunction theorem}

$(p \wedge q \wedge r)$ *unless* $((p \wedge q \wedge \neg q) \vee (q \wedge r \wedge \neg p) \vee (\neg p \wedge \neg q))$

$=$ {Rewriting the right side}

$(p \wedge q \wedge r)$ *unless* $((r \vee \neg q) \wedge \neg p)$

$\Rightarrow$ {Consequence Weakening}

$(p \wedge q \wedge r)$ *unless* $\neg p$ (14)

The conjunction of 13 and 14 is exactly the definition of $p \mapsto\!\!\!\!\!\!\cdot\;\; (q \wedge r)$, which concludes the theorem. □

Theorem A.5 *Restricted Consequence Weakening:*

$$\frac{(p \mapsto\!\!\!\!\!\!\cdot\;\; (q \wedge r))\ ,\ (p \Rightarrow r)}{p \mapsto\!\!\!\!\!\!\cdot\;\; q}$$

Proof:

$p \mapsto\!\!\!\!\!\!\cdot\;\; (q \wedge r)$

$\Rightarrow$ {Definition of $\mapsto\!\!\!\!\!\!\cdot$}

$p \mapsto ((q \wedge r) \vee \neg p)$

$\Rightarrow$ {Consequence Weakening}

$p \mapsto (q \vee \neg p)$ (15)

$(p \mapsto\!\!\!\!\!\!\cdot\;\; (q \wedge r)) \wedge (p \Rightarrow r)$

$\Rightarrow$ {Definition of $\mapsto\!\!\!\!\!\!\cdot$}

$((p \wedge q \wedge r)$ *unless* $\neg p) \wedge (p \Rightarrow r)$

$\Rightarrow$ {Rewriting the left side}

$(p \wedge q)$ *unless* $\neg p$ (16)

The conjunction of 15 and 16 is exactly the definition of $p \mapsto\!\!\!\!\!\!\cdot\;\; q$, which concludes the theorem. □

Theorem A.6 *Precondition Strengthening:*

$$\frac{(p \Rightarrow q)\ ,\ (q \mapsto\!\!\!\!\!\!\cdot\;\; r)}{p \mapsto\!\!\!\!\!\!\cdot\;\; r}$$

Proof:

$$
\begin{array}{ll}
 & (p \Rightarrow q) \wedge (q \mapsto\!\!\!\!\!\!\cdot\;\; r) \\
= & \quad \{\text{Implication theorem}\} \\
 & (p \Rightarrow q) \wedge (p \mapsto\!\!\!\!\!\!\cdot\;\; q) \wedge (q \mapsto\!\!\!\!\!\!\cdot\;\; r) \\
\Rightarrow & \quad \{\text{Weak Transitivity theorem}\} \\
 & (p \Rightarrow q) \wedge (p \mapsto\!\!\!\!\!\!\cdot\;\; (q \wedge r)) \\
\Rightarrow & \quad \{\text{Restricted Consequence Weakening}\} \\
 & p \mapsto\!\!\!\!\!\!\cdot\;\; r
\end{array}
$$

□

Theorem A.7 *Conjunction:*

$$\frac{(p \mapsto\!\!\!\!\!\!\cdot\;\; q)\ ,\ (p' \mapsto\!\!\!\!\!\!\cdot\;\; q')}{(p \wedge p') \mapsto\!\!\!\!\!\!\cdot\;\; (q \wedge q')}$$

Proof:

$$
\begin{array}{ll}
 & (p \mapsto\!\!\!\!\!\!\cdot\;\; q) \wedge (p' \mapsto\!\!\!\!\!\!\cdot\;\; q') \\
\Rightarrow & \quad \{\text{Definition of } \mapsto\!\!\!\!\!\!\cdot\;\;\} \\
 & (p \mapsto ((q \wedge p) \vee \neg p)) \wedge (p' \mapsto ((q' \wedge p') \vee \neg p')) \qquad (17)
\end{array}
$$

$$
\begin{array}{ll}
 & (p \mapsto\!\!\!\!\!\!\cdot\;\; q) \wedge (p' \mapsto\!\!\!\!\!\!\cdot\;\; q') \\
\Rightarrow & \quad \{\text{Definition of } \mapsto\!\!\!\!\!\!\cdot\;\;\} \\
 & ((p \wedge q)\ \textit{unless}\ \neg p) \wedge ((p' \wedge q')\ \textit{unless}\ \neg p') \qquad (18)
\end{array}
$$

$$
\begin{array}{ll}
 & (p \mapsto\!\!\!\!\!\!\cdot\;\; q) \wedge (p' \mapsto\!\!\!\!\!\!\cdot\;\; q') \\
\Rightarrow & \quad \{17 \text{ and } 18\} \\
 & (p \mapsto ((q \wedge p) \vee \neg p)) \wedge (p' \mapsto ((q' \wedge p') \vee \neg p')) \wedge \\
 & ((p \wedge q)\ \textit{unless}\ \neg p) \wedge ((p' \wedge q')\ \textit{unless}\ \neg p') \\
\Rightarrow & \quad \{\text{Generalization of the Completion Theorem}\} \\
 & (p \wedge p') \mapsto (((p \wedge q) \wedge (p' \wedge q')) \vee \neg p \vee \neg p') \\
= & \quad \{\text{Rewriting the right side}\} \\
 & (p \wedge p') \mapsto ((q \wedge q') \vee \neg(p \wedge p')) \qquad (19)
\end{array}
$$

$$
\begin{array}{ll}
 & (p \mapsto\!\!\!\!\!\!\cdot\;\; q) \wedge (p' \mapsto\!\!\!\!\!\!\cdot\;\; q') \\
\Rightarrow & \quad \{\text{Definition of } \mapsto\!\!\!\!\!\!\cdot\;\;\} \\
 & ((p \wedge q)\ \textit{unless}\ \neg p) \wedge ((p' \wedge q')\ \textit{unless}\ \neg p') \\
\Rightarrow & \quad \{\text{Simple Conjunction}\} \\
 & (p \wedge q \wedge p' \wedge q')\ \textit{unless}\ \neg p \vee \neg p' \\
= & \quad \{\text{Rewriting}\} \\
 & ((p \wedge p') \wedge (q \wedge q'))\ \textit{unless}\ \neg(p \wedge p') \qquad (20)
\end{array}
$$

The conjunction of 19 and 20 is exactly the definition of $(p \wedge p') \mapsto\!\!\!\!\!\!\cdot\;\; (q \wedge q')$, which concludes the theorem.

□

Theorem A.8 *Generalization of the Conjunction theorem:*
For any finite set of predicates p_i, q_i, where $0 \leq i < N$:

$$\frac{\langle \forall i \;::\; p_i \mapsto\!\!\!\!\!\!\cdot\;\; q_i \rangle}{\langle \forall i \;::\; p_i \rangle \mapsto\!\!\!\!\!\!\cdot\; \langle \forall i \;::\; q_i \rangle}$$

Proof: The result is trivial for $N \leq 1$. For $N = 2$, it follows directly from the Conjunction theorem. Use induction over N.
□

Theorem A.9 *Stability:*

$$\frac{(p \mapsto\!\!\!\!\!\!\cdot\;\; q)\ ,\ (\mathit{stable}\ p)}{(p \mapsto q), (\mathit{stable}\ (p \wedge q))}$$

Proof:

$(p \mapsto\!\!\!\!\!\!\cdot\;\; q) \wedge (\mathit{stable}\ p)$
$\Rightarrow$ {Definition of $\mapsto\!\!\!\!\!\!\cdot$}
$(p \mapsto (q \vee \neg p)) \wedge (\mathit{stable}\ p)$
$\Rightarrow$ {Definition of *stable* p}
$(p \mapsto (q \vee \neg p)) \wedge (p\ \mathit{unless}\ \text{false})$
$\Rightarrow$ {PSP theorem}
$p \mapsto ((q \vee \neg p) \wedge p)$
$\Rightarrow$ {Rewrite the right side, Consequence Weakening}
$p \mapsto q$ (21)

$(p \mapsto\!\!\!\!\!\!\cdot\;\; q) \wedge (\mathit{stable}\ p)$
$\Rightarrow$ {Definition of $\mapsto\!\!\!\!\!\!\cdot$}
$((p \wedge q)\ \mathit{unless}\ \neg p) \wedge (\mathit{stable}\ p)$
$\Rightarrow$ {Definition of *stable* }
$((p \wedge q)\ \mathit{unless}\ \neg p) \wedge (p\ \mathit{unless}\ \text{false})$
$\Rightarrow$ {Conjunction, and rewriting}
$(p \wedge q)\ \mathit{unless}\ \text{false}$ (22)

The conjunction of 21 and 22 concludes the theorem.
□