Ada/TL Specification and Verification of a Distributed Computation

William Hankley & Peikun Tsai

Kansas State University

Abstract

Ada/TL is a specification language that combines aspects of Ada, VDM, and temporal logic. It is styled to provide a bridge between techniques for engineering of distributed software and techniques for formal specification and verification. An Ada/TL specification builds on an Ada specification of the structure of task interfaces; semantics of task operations are given by assertions using abstract structures of VDM; the behavior of each task is given by a temporal assertion about the task operations; finally, global properties of the network are given by a temporal predicate about the interaction of tasks. Verification of the the specification consists of a demonstration that the system global properties follow from the rest of the specification. Specifications are constructive in that the verified specification can be transformed to a distributed Ada program. This paper illustrates the methodology by first specifying a distributed computation for summing numbers stored at nodes of a network and second showing a proof that the computation is correct. The proof illustrates the common technique of induction over the nodes of the network. The structure of this example suggests an approach for correctly specifying other distributed computations.

Background

In spite of some weaknesses, Ada has been promoted as a foundation for specification languages for distributed systems [4]. Some specification languages build upon the Ada framework, for example ANNA [12], TSL [11], and Ada/TL [5]. One advantage of building upon the Ada base is that the specification starts with a style and notation that is used by software engineers. For concurrent and distributed systems a further advantage is that an Ada based specification defines the highest level of tasking structure for the target system; there is not a drastic change of structure from specification to software design.

Earlier papers [6,13] have illustrated Ada/TL specification and proof techniques for concurrent systems. These are summarized next.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1992 ACM 0-89791-502-X/92/0002/0846...$1.50

The Ada/TL language starts with the Ada specification part as the specification framework. It adds assertions to define the semantics of task operations. It uses abstract data structures and notations from the Vienna Development Method (VDM [8]) to model information within tasks and to express task assertions. Experiences with VDM (and similar results with Z) show that specifications expressed using abstract data structures are more expressive and more concise than concrete code, yet they are easily translated to concrete code. Finally, Ada/TL uses temporal logic to define temporal behavior of tasks. Within each task, a temporal assertion defines the direct interaction of the task with other tasks. Temporal predicates can be interpreted as defining states and transitions. Lamport [10] argues that is a natural approach which allows easy understanding of such specification. At the same time, temporal predicates rest on the foundation of formal logic [9] and thus allow formal verification of specifications. In addition to the task assertions, Ada/TL system-wide temporal assertions state expected safety and liveness properties. These assertions always state expected results of indirect interactions of tasks. If the system assertions do indeed follow from the task specifications, then the system assertions can be verified using formal proof techniques.

For purposes of verification, the temporal behavior defined by each task properly can be seen as an I/O automaton. While the composite system automaton has many states, there are typically few composite states of allowed task interaction. Verification of the global property is achieved by symbolically tracing and justifying global predicates over all branches between states of allowed task interaction. For continuing tasks, there are always recurring states. Verification requires an induction over the recurring states much like the inductive proof of loop properties for sequential programs. Of course, in an environment of software development, formal proofs may give way to less formal reviews of system properties; yet, understanding of the underlying proof structure is essential for understanding correctness of specified properties.

Motivation

This paper extends the Ada/TL methodology to the case of a computation distributed over a network of tasks. Two concepts arise in this paper, first, the Ada specification of distributed program components and second, the proof technique itself. These issues are discussed separately below and then are illustrated using a distributed algorithm for summing numbers stored a sties of a network. The example is the focus of the body of the paper. The example is a finite (terminating) computation, but it is distributed over an arbitrary number of network nodes. The algorithm is a kind of "diffusing computation" algorithm [3]. The algorithm is important in that other kinds of "diffusing computation" algorithms can be reduced to this form [7].

The Ada language represents a system of tasks as a single concurrent program. There are no special Ada language features for distributed programs. A number of authors [1, 2, 14] have presented ways for extending Ada for distributed applications. The prevailing approach is to break a distributed system into separate programs each allocated to a network node. Communication between separate nodes may be supported by a distributed kernel using mechanisms such as remote rendezvous, remote procedure calls, asynchronous message passing, or message broadcasting.

For purposes of defining global properties, Ada/TL treats a distributed system as a single integrated specification with a family of packages each assigned to a network node. Global variables of the specification can be used to state requirements and constraints, but they cannot be used to define the computation part of the specification.

In earlier papers, verification of a properties of a system of tasks required inductive proof of properties at recurring states of the computation. In the example of this paper, verification of the system property does not require induction over recurring states, but it does require induction over the number of nodes. The proof is made tractable by decomposing it into a hierarchy of related subproperties, which are presented as lemmas. The lemmas allow the proof to be stated in concise steps.

The distributed algorithm and underlying concept of the proof are not new. The contribution of this presentation is to show the specification and proof in a notation and style which couples directly with Ada notation and style. We believe that this style of specification & verification can be used by practising software engineers.

Statement of the problem and the algorithm and explanation of the Ada/TL notation are given in the next section and then verification of the specification is presented.

Specification of Requirements

The network consists of an arbitrary number of sites each labeled with a unique siteId identifier called "MyId". Each site supports communication with a subset of neighbor sites using remote rendezvous. Each site knows the siteId identification of its neighbor sites. The network is connected in that there is a path of communication between any two sites, possibly via intermediate sites. Each site stores information, which is specified as a positive integer called "No(i)" where "i" is a siteId. A single site, identified as "Root", seeks to find the sum of all numbers stored at all sites of the network.

Specification of the problem constraints is given in Figure 1. The whole network is the package "Network". It is not installed at any site, but it is the global view of the network. The global variables "Edges", "Root", and "Nums" are not stored at any of the sites, but they represent the global definition of the network. The various abstract structures (sets, maps) are not part of Ada but are taken from VDM. The configuration of the network is defined by the structure "Edges", which is a set of 2-tuples of node identifiers.

The " - - |" lines mark several kinds of assertions and definitions, using the following key words:

"req" assertions are required conditions which the subject structures must meet, so these are not subject to any verification;
"def" lines define functions which are used in assertions but not directly used within the computation defined by the specification;
"init" assertions define initial states; "sys property" is the global property to be verified;
"in" and "out" assertions define conditions for variables and parameters at the beginning and ending points of task operations (none of these are used in Figure 1);
"inv" are invariant properties of variables.

Constrained sets are defined using a set constructor with the form: set(<bound_var> | <binding__exp>)
This forms the set of elements that are formed from components of some other structure and which satisfy the constraint expression .

Predicates over structures have the general form:
(<quantifier> <binding_exp> : <*boolean*_exp>)
with quantifiers such as "all" and "exist". A summing function has a similar form, but it using the quantifier "sum".

The generic package "SitePkg" is to be instantiated at each site. Each site must define the parameters of "MyId" which is the siteId, "Nbs" which is the set of neighbor siteId's, and "N" which is the number stored at the site. A task "Init" is defined for the site "Root" with the understanding that "Init" will start the computation.

The only temporal predicate is the system property which states that eventually (the $\Diamond$ operator) the variable "S" at the "Root" site determines the sum of "Nums" values stored over the network.

Specification of the Computation

A naive approach for computing the required sum is for site "Root" to query all other sites with a broadcast message, receive as response the stored value from each site, and then compute the sum. A more distributed form of the algorithm is the following:

i) Site "Root" queries its neighbor.
ii) Each site which receives a query in turn queries its other neighbor sites.
iii) When a site first receives a query, it records the calling site as its "ParentId" and later reports back its partial sum. When a site receives any other query, it immediately reports back the value zero.
iv) When each site receives a return report of a partial sum, it records the partial sum value. When the site has received a return report from all of its neighbors (excluding its "ParentId "site), it is then able to sum the stored values and report its partial sum as stated in item (iii).

The specification for the distributed algorithm is given in Figure 2. Line labels are included for reference in the verification. For each site package, a task "Sum" is defined with communication ports "Start" and "Ack". "Start" receives requests to compute a partial sum and "Ack" receives the corresponding reports of the computed partial sums. The functional behavior of task operations is defined by their "in" and "out" assertions; the temporal interaction of the task operations is defined by the temporal "property" assertion. The entry assertions for "Start" indicate that for the first call the variables "ParentId" and "Np" are bound. "Np" is the set of neighbors excluding the calling parent site. For subsequent calls to "Start" the "StartCount" is bound to one more than the previous value. The entry assertions for "Ack" indicate that the returned sum "Partial" is bound as the "RId" component of "Nbs".

The following temporal operators are used to form temporal predicates:

$\Diamond$ P == eventually P is true
seq(P1,P2) == P1 is true before P2
seq(P1, P2, ...) == P1 is true before seq(P2, ...)
$\Diamond^{c}$ P == (c and P) is true zero or more times in sequence until eventually c is false

The "property" temporal predicates are explained below as sequential behavior of the task.

The property for task "Sum" indicates that it first accepts a "Start" request and then propagates (or diffuses) the request to all of its other neighbor sites (determined by "Np"). Thereafter, it asynchronously accepts either "Start" or "Ack" entries so long as there are remaining operations to be processed, as indicated by the condition "StartCount < |Np| or S = 0 " . Whenever an additional "Start" is received the value "0" is immediately returned. When the last "Ack" is received (indicated by "Npa = Np") then the computed partial sum "S" is returned to the "ParentId" task.

This kind of distributed algorithm could hardly be created without understanding the underlying structure which allows correctness to be verified. The key ideas are illustrated for a typical site "i" shown in Figure 3. "Start" requests dynamically form a spanning tree of sites over the whole network. The spanning tree is not unique and is not statically determined; it depends in part on the transit times of the "Start" requests. At each site, the first "Start" request determines the parent for the site. For site "i", the neighbors for which "i" becomes the parent form the "Children(i)". Other neighbors of site "i" form "Others(i)". The spanning tree is formed only from the "Children" for each site. Each site of the spanning tree computes a partial sum only for its "Children" sites and it reports back the partial sum only to its parent site. Sites of "Others(i)" (which have some "parentId" other than "i") always report back "0" to site "i". The global concepts of kinds of nodes (parents, children, and other) and the spanning tree are included as part of the specification.

The boolean function "SP(i)" is used to form the global system property. "SP(i)" states that for task "Sum(i)" when its parent task "T(p)" issues a "Start" request then eventually "Sum(i)" will form the partial sum of all numbers stored in its subtree (including its own number) and report back the sum to its parent site. Task "T(p) "could be either "Init(Root)" or some other "Sum(ParentId)".

The global "sys property" states that "SP(i)" holds for all sites and eventually variable "S" of the "Init" task is bound as the sum of all the numbers stored in the network.

Verification of Distributed Summing

In the network specification, the "in" and "out" assertions define the entry operations and "property" assertions define the sequential behavior of each task. These are taken as premises for the verification. The objective is to show that the system property holds. Verification proceeds in two steps. First we show that the predicate "SP(i)" holds for every site "i". That consists of showing that "SP(i)" holds directly for every leaf site and then showing by induction that "SP(i)" holds for arbitrary sites. Then, the correctness condition follows from "SP(Root)" and the condition that "Subtree(Root) = Connect(Root)". To simplify some steps of the proof, various lemmas are used. The organization of the subproofs is shown in Figure 4. The proofs listed in the Appendix.

In the general case, the steps of the proof will symbollicaly trace the interaction of tasks, where the behavior of each task is given by its "property" assertion. For this simple example, the paths consist of only threads thru tasks "Sum(i)". In reading the verification, we will stand in the position of a task "Sum(i)" at site "i". This means that any variable "X" stands for "Sum(i).X" .

Referring to the Appendix, the steps are labeled first with a step number of the form n-m, corresponding to the n th step of a path and m th branch within a path. Branches labeled as n' or n-m' do not lead to the final step of the proof; either the branch is excluded by subproof conditions or the branch leads back to a repeating state which terminates with some "eventually" condition. The predicate for each step either is taken directly from the specification line number in the second column or it is derived by logical inference as explained by the comment in the right hand column. Inference steps are presented in a semi-formal manner which is intended to suited for peer review. Within the proof, certain operators such as "all" and "set" are replaced with their equivalent symbolic form.

Based on Figure 4, the verification starts from the bottom of the hierarchy with Lemma 1. Lemma 1 states that if site "i" is a site in "Others(p)" of another site "p", then after site "p" sends "Start(p)" to "i", "Sum(i)" will finally return "Ack(i,0)" to "Sum(p)". When the neighbor site "p" sends "Start" to "i", it may be received with either of two different "Start" conditions, p1 or p4 . Since "i" belongs to "Others(p)", "Start(p)" must be received through condition p4, and p4 can only be accessed when condition p3 is true. At the moment the checking p3, it must be true that "StartCount < |Np|" since at least one "Start" (namely "Start(p)") has not been received yet. From p5, "Sum(i)" returns "Ack(i, 0)" to "Sum(p)". Thus, Lemma 1 holds.

Lemmas 1a, 1b, and 1c follow in a similar way. Details are not shown.

The SP:Non-recursive part of "SP" states that for a leaf site "i" "SP(i)" is true. After establishing the "ParentId" and "Np" set, there are two main cases to be considered, labeled as branches *-1 and *-2 . Both cases must lead to the conclusion part of "SP(i)". In both cases, since "i" is a leaf site, the sum of numbers over the "Subtree(i)" is merely the number "N" stored at site "i". For the "1" branch site "p" is the only neighbor of "i" and the result follows directly. For case "2", site "i" propagates other "Start" requests (line 3-2) but those all must respond with return value "0" since "i" is a leaf. In line 5-2', site "i" may receive additional "Start" requests, but those do not effect the "SP" result. Obviously, the operations of concern are "Ack" in line 5-2. Eventually all member of "Np" respond , so that "Npa = Np". Line 8-2' indicates further "Ack" responses are due, which leads back to line 5-2. When "Npa=Np" (line 8-2) , the conclusion follows easily.

The SP:Recursive part generalizes the SP:Non-recursive part. The major difference is that "Children(i)" is not empty, so "Np" is the union of "Children(i)" and "Others(i)". Lemma 2 was stated to ensure that for all "j" in "Children(i)", once

"Start(i)" is sent to "Sum(j)" , then "Sum(i)" will finally receive an "Ack" from j. Lemmas 1 and 2 together tells us all neighbor sites of "i" (excluding its parent) will finally return an "Ack". The induction hypothesis for the recursive proof is that "SP(j)" is true for all "j" in "Children(i)", and the goal to be proved is that "SP(i)" is true. Since we assume there are no communication failures, Lemma 2 follows from the induction hypothesis. Step 5 generates the major result. In this step, site "i" has received all "Ack" responses. Since each received number is the sum over a subtree, then computed result "S" is shown to be equal to a sum over the "Subtree(i)".

Since "SP(i)" must hold for "i=Root", the desired "sys property" will follow provided that "Subtree(Root) = Sites", which is stated as Lemma 3. Step 1 of Lemma 3 follows easily. Step 2 is by contradiction. If there is a site α that is not part of the "Subtree(Root)" then α violates the "Connected" assumption.

Conclusion

We have presented a conceptually simple structure for formalizing and verifying a representative algorithm for diffusing computations. The site and task structures follow closely the form of Ada specifications. Within the specification, task properties define allowed sequences of states of the task. The specification must also include the underlying definitions and invariants that allow the algorithm to be understood and to be verified. For this simple algorithm, the verification follows naturally as symbolic evaluation of state predicates along allowed behavior paths. Because the proof is annotated with specification line numbers and explanations of inference steps, the proof should be easy to explain to peer reviewers. For more complex algorithms, it is likely that proofs would require further relating threads of behavior of multiple tasks, which is not treated in summing proof.

References

[1] C. Atkinson, T. Moreton, A. Natali, *Ada for Distributed Systems*, Cambridge University Press, 1988.

[2] D. Bhatt, "Implementing a Distributed Fault-Tolerant Embedded System in Ada", *Proceedings of Tri-Ada 90 Conference*, ACM, Dec 1990, pp 323-331..

[3] E. W. Dijkstra and C. S. Scholten, "Termination Detection for Diffusing Computations", *Information Processing Letters*, Vol. 11, No 1, 1980, pp 1-4.

[4] S. J. Goldsack, *Ada for Specification: Possibilities and Limitations*, Cambrige University Press, 1985.

[5] W. Hankley and J. Peters, "Temporal Specification of Ada Tasking", *Proceedings of the 23th Hawaii International Conference on System Sciences*, Vol II, IEEE, Jan 1990, pp 410-419.

[6] W. Hankley and J. Peters, "A Proof Method for Ada/TL", *Proceedings of the 8th Annual National Conference on Ada Technology*, Mar 1990, pp 392-398.

[7] M. A. Huang and S. Teng, "Secure and Verifiable Schemes for Election and General Distributed Computing Problems", *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, Aug 1988, pp 182-196.

[8] C. B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall, 1990.

[9] F. Kroger, *Temporal Logic of Programs*, Springer-Verlag, 1987, pp 148.

[10] L. Lamport, "A Simple Approach to Specifying Concurrent Systems", *Comm. ACM*, Jan 1989, pp 32-46.

[11] D. Luckham et al., "Task Sequencing Language for Specifying Distributed Ada Systems", *Lecture Notes in Computer Science* No. 275, Springer-Verlag, 1987, pp 249-305.

[12] D. Luckham, *Programming with Specifications: An Introduction to Anna, A Language for Specifying Ada Programs*, Springer-Verlag,1990

[13] J. Peters and W. Hankley, "Proving Specifications of Tasking Systems Using Ada/TL", *Proceedings of Tri-Ada 90 Conference*, ACM, Dec 1990, pp 4-13.

[14] D. Wengelin and L. Asplund, "Application of Ada on a Distributed Missile Control System", *Proceedings of Tri-Ada 90 Conf.*, ACM, Dec. 1990, pp 300-313.

```
-- Figure 1

package Network is

type SiteIdType is pending;                                 -- structure of site Id's left unspecified
                                                            -- global variables used to define the network
Edges: set of (SiteIdType, SiteIdType) := pending;          -- set of 2-tuples which defines the network
                                                            -- actual value of Edges is not specified
--| req  (all (s1,s2)  in Edges:  (s2, s1) in Edges);        --        Edges is symmetric
--| req  not (exist (s1,s1) in Edges ) ;                     --        no self-edges

Sites : set of SiteIdType;                                  --  set of all site Id's
--| req  Sites = set( s | (s, _) in Edges) ;                --  Sites is defined from Edges

Root: SiteIdType;
--| req  Root in Sites;                                     --  Root is a particular Site
--| def  Nbs(i: SiteIdType) == set( s |  (i,s) in Edges);   --  set of neighbors is defined by Edges
--| def  Connect(i : SiteIdType) == set(i) +                --  set of all sites directly or indirectly
             set( s |  s in Connect(s1) and s1 in Nbs(i) ); --    connected to some site i
--| req  Sites = Connect(Root);                             --  all sites are connected to site Root

Nums:  map SiteId to Positive := pending;                   --  number stored at each site, unspecified

generic   MyId: SiteIdType;                                 --  parameters for package to be installed
          Nbs: NbSet;                                       --    at each site
          N: Positive;

package SitePkg   is
--| req   N = Nums(MyId);

     S:  positive;                                          -- computed sum information

     task Init(Root) is                                     -- initiation task only for site Root
     --| req   MyId = Root;
     end task Init;

--| init  Init(Root);                                       -- initial condition: activates task Init(Root)

end package SitePkg;

--| req (all i in Sites:  Site(i) is SitePkg(i, Nbs(i) , Nums(i)) );   -- SitePkg is instantiated at each site
--| sys property
        ◇  SitePkg(Root).S = (sum i in Sites:  Nums(i)  ) ;  -- eventually, S at the Root site is the
                                                            --   sum of all No's stored at all sites

end package Network;                          --  end of package structure;
```

Figure 1.   Specification of Requirements

```
- -  start of Figure 2.  Specification of Distributed Computation

package Network is
- -  SiteIdType,  Edges, Sites, Root, Nbs(i), Nums(i)  as in version 1 ,  see Figure 1;

generic   MyId: SiteIdType; Nbs: NbSet; N: Positive;                   - -  parameters for package

package SitePkg    is

      task Sum(MyId) is                                                - - MyId is the siteId,
            NbSum:  map SiteIdType to Natural := null;                 - -    only one Sum task at each site
            S: positive := 0 ;
            ParentId := SiteIdType := null;
            StartCount : Natural := 1;

            entry Start( PId: SiteIdType);                             - - request from site which may be the parent
            - - | in PId in Nbs(MyId);
s1          - - | out if ParentId=null then
s2                    Np = Nbs -  PId
s3                    and ParentId = PId                               - - PId was the parent siteId
s4                    and  if Np = null then
s5                        S'out = N
s6                        and Sum(ParentId).Ack(MyId, S)
                          endif
                     else
s7                    StartCount'out = StartCount'in + 1               - - PId was not the parent siteId
                     endif;

            entry Ack(  RId: SiteIdType;  Partial: Natural);           - - RId = responder's siteId
            - - |  in   RId in Nbs;
a1          - - | out  NbSum(RId) = Partial
a2                 and  Npa'out = Npa'in + RId;
            - - | property
p1                seq( Start(PId),                                     - - PId is the parentId
p2                     (all j in Np:  Sum(j).Start(MyId) ),
p3                    ◇  (StartCount < |Np|  or  S = 0)
p4                     (seq( Start(XId),                               - - XId in Others(MyId)
p5                            Sum(XId).Ack(MyId, 0) ) )
                        or
p6                      seq(Ack(RId, Partial),
p7                           if Npa = Np then                          - - all Start requests have responded
p8                              S = (sum s in Np: NbSum(s) ) + N  endif,
p9                           if MyId = Root
p10                             then  Init(Root).Ack(Root, S)
p11                             else   Sum(ParentId).Ack(MyId, S) endif)
                     );
      end task Sum(MyId);

      task Init(Root) is                                               - -  initiation task only for site Root
      - - | req   MyId = Root;
i1    - -  | property seq( Sum(Root).Start(Root),
i2                        Ack(Root, S) );
      end task Init;
- - | init  Init(Root);                                                - -  initial condition: activates task Init(Root)

- -  following structures are basis for design and verification of the algorithm
- - | def    Np(i: SiteIdType) ==  Nbs(i) - set( Sum(i).ParentId) ;
- - | def    Children(i: SiteIdType) == set( x | x in Np(i)  and  ◇  (Sum(x).ParentId =  i ) );
- - | def    Others(i: SiteIdType)  = = set( x | x in Np(i) and  ◇  (Sum(x).ParentId ≠i ) );
- - | inv    ( all i in Sites:  Np(i) = Children(i) + Others(i) );         - - clear from definitions above
- - | def    Leaf(i: SiteIdType) == (Children(i) = null );
- - | def    Subtree(i: SiteIdType) == set( i) +  set(x | s in Children(x) and x in Subtree(s));
end package SitePkg;                                                   - -  Figure 2 continued on next page
```

```
- -  continuation of Figure 2

- - | req (all s in Sites:  Site(s) is SitePkg(s, Nbs(s) , No(s)) );         - - SitePkg is instantiated at each site

- - | def SP(i: SiteIdType) == (exist T(p) is task :
                                  (T(p):Sum(i).Start(p) and Sum(i).ParentId = p)
                                   imp ◇  (Sum(i).S = (sum s in  Subtree(i): Sum(s).N )
                                              and  Sum(i): T(p).Ack(i, Sum(i).S) )
                                 );
- - | sys property
            (all i in Sites: SP(i) )
          and ◇  Init(Root).S = (sum i in Sites:  Numsi)  ) ;           - - eventually, S at the Root site is the
                                                                       - -  sum of all No's stored at all sites
end package Network;                                                  - -  end of package structure;
```

Figure 2. Specification of Distributed Computation

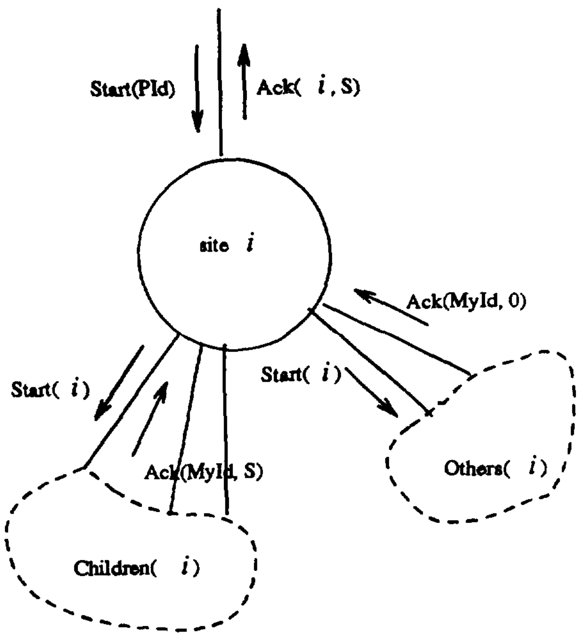

Figure 3. Children and Others for a Certain Site i

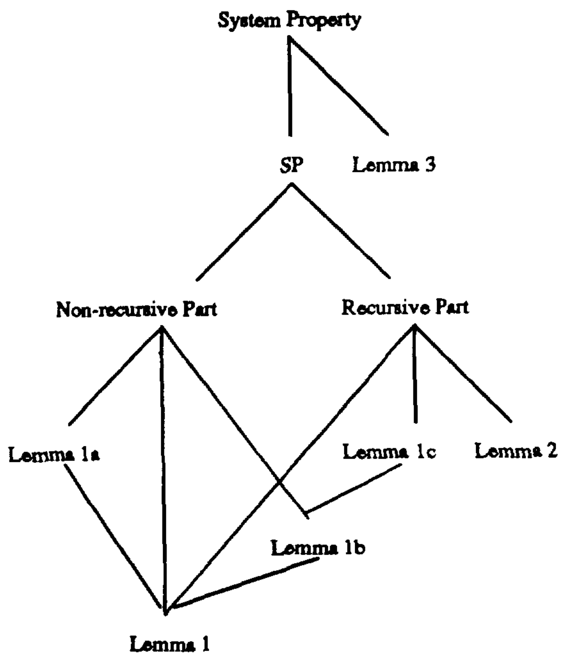

Figure 4. Hierarchy of Proof Structure

Appendix

Lemma 1:

$\forall$ p∈Sites:($\forall$ i∈Others(p): Sum(i).Start(p)→ ◇ Sum(p).Ack(i, 0))

```
        Since i ∈ Others(p), following predicate holds for site i:
0       ParentId ≠ q;                          //def of Others(p)
        StartCount < |Nbs|;                    //not rec Start(p) yet

1   p3  $\diamond^{(StartCount<|Nbs| \text{ or } S=0)}$   //fr 0
2   p4  case Start(Xid)                        //rec Start fr Xid≠ParentId
3'          case Xid ≠ p                       //don't care about this case
3           case Xid = p
    s1          (ParentId ≠ null)              //fr 0
    s7          StartCount'out = StartCount + 1
4   p5          Sum(Xid).Ack(MyId, 0)          //MyId=i by assumption
                ⇒Sum(p).Ack(i, 0)              //fr 3 case, MyId=i
5               Lemma 1=true                   //fr 4
2'  p4  case Ack(RId, Partial)                 //don't care about this case
```

Lemma 1a:

$\forall$ p∈Sites:($\forall$ i∈Others(p): Sum(i).Start(p)→ ◇ NbSum(i)=0)
Lemma 1a can be directly proved by applying Lemma 1 and Ack:a1.

Lemma 1b:

$\forall$ p∈Sites:($\forall$ i∈Others(p): Sum(i).Start(p)→ ◇ Others(p)⊂Npa(p))
Lemma 1b can be directly proved by applying Lemma 1 and Ack:a2.

Lemma 1c:

$\forall$ p∈Sites:Leaf(p) and ($\forall$ i∈Others(p): Sum(i).Start(p))→ ◇ (Others(p)=Npa(p))
Lemma 1c can be directly proved by applying Lemma 1b with Children(p)=ϕ.

SP : Non − recursive Part: A leaf site i satisfies SP(i).

$\exists$ p∈Sites:T(p):(leaf(i) and T:Sum(i).Start(p) and Sum(i).ParentId=p
→ ◇ Sum(i).S = (sum s∈Subtree(i): Sum(s).N) and T:Ack(i, Sum(i).S))

```
0       S = 0; StartCount = 1; ParentId = null;
                Np = ϕ; Npa = ϕ;               //fr var initiation
1   p1  Start(Pid)                             //rec Start fr p, Pid=p
    s1  (ParentId = null)                      //fr 0
    s2  Np = Nbs - Pid
    s3  ParentId = Pid
2-1 s4  case Np = ϕ                            //i's only neighbor is Pid
    s5      S = N
            ⇒S = (sum s∈Subtree:Sum(s).N)      //inv of Leaf
    s6      Sum(ParentId).Ack(MyId, S)
            ⇒ Sum(p).Ack(i, S)                 //1.p1&1.s3, MyId=i
3-1 p2  ∀ j∈Np:Sum(j).Start(i)
        ⇒ ∀ j∈ϕ:Sum(j).Start(i)                //case 2-1
        ⇒ true                                 //empty domain
4-1 p3  $\diamond^{(StartCount<|Nbs| \text{ or } S=0)}$
        ⇒false                                 //StartCount=|Nbs|&S=N
5-1     SP(i)=true                             //fr 2-1.s5&2-1.s6
2-2 s4  case Np ≠ ϕ                            //i has more than one neighbors
            ⇒ true                             //no assertions
```

```
3-2    p2    ∀ j∈Np:Sum(j).Start(i)
             ⇒ ∀ j∈Others:Sum(j).Start(i)                   //Children=φ
             ⇒ ◇∀ j∈Others:Sum(i).Ack(j, 0)                 //by Lemma 1
4-2          $\diamond^{(StartCount<|Nbs|\ or\ S=0)}$         //initially S=0
5-2'   p4    case Start(Xid)                                //don't care about this case
5-2    p6    case Ack(RId, Partial)                         //rec Ack fr 3-2
                Partial = 0; RId∈Others                     //var matching
       a1       NbSum(RId) = Partial                        //out of Ack
                ⇒NbSum(RId) = 0                             //fr 5-2.p6
       a2       Npa'out = Npa + RId                         //out of Ack
6-2             ◇ ( (sum s∈Others: NbSum(s))=0 )            //Lemma 1a
7-2             ◇ (Npa = Others)                            //Lemma 1c
                ⇒ ◇ (Npa = Np)                              //Children=φ
8-2'   p7       case Npa ≠ Np
                ⇒ true                                      //no assertions
8-2    p7       case Npa = Np                               //◇ true fr 7-2
       p8          S = (sum s∈Np:NbSum(s)) + N
                   ⇒S = (sum s1∈Others:NbSum(s1)) +
                      (sum s2∈Children:NbSum(s2)) + N       //inv of Np
                   ⇒S = 0 + 0 + N                           //6-2,Children=φ
                   ⇒S = N                                   //algebra
                   ⇒S = (sum s∈{i}: Sum(s).N)               //N≡Sum(i).N
                   ⇒S = (sum s∈Subtree: Sum(s).N)           //inv of Leaf
9-2    p9          MyId ≠ Root                              //not Root
       p11         Sum(ParentId).Ack(MyId, Sum(i).S)
10-2               SP(i) = true                             //fr 8-2.p8&9-2.p11
```

<u>*end of SP : Non − recursive Part*</u>

<u>*SP : Recursive Part*</u>: Let i be an internal node, and assume
(∀ j∈Children(i): SP(j)=true). We want to prove SP(i)=true.

<u>*Lemma 2*</u>:

∀ i∈Sites: (∀ j∈Children(i): Sum(j).Start(i))→ ◇ Sum(i).Ack(j, Sum(j).S)
Lemma 2 holds following from assuming for all j∈Children(i) SP(j) is correct
By attaching Ack:a2 to SP(j), we know that Sum(i) will finally receive
Ack(j, Sum(j).S) from site j.

```
0            S = 0; StartCount = 1; ParentId = null;
                   Np = φ; Npa = φ;                         //fr var initiation
1      p1    Start(Pid)                                     //rec Start fr p, Pid=p
       s1    (ParentId = null)                              //fr 0
       s2    Np = Nbs - Pid
       s3    ParentId = Pid
       s4    Np ≠ φ                                         //i is not a leaf
             ⇒ true                                         //no assertions
2      p2    ∀ j∈Np:Sum(j).Start(i)
             ⇒ ◇(∀ j1∈Children:Sum(i).Ack(j1, Sum(j1).S)
                and ∀ j2∈Others:Sum(i).Ack(j2, 0))          //Lemma 2 & Lemma 1
3      p3    $\diamond^{(StartCount<|Nbs|\ or\ S=0)}$         //StartCount< |Nbs|, S=0
4'     p4    case Start(Xid)                                //don't care about this case
4      p6    case Ack(RId, Partial)
                if RId ∈ Children then Ack(RId, Sum(RId).S)
                   else Ack(RId, 0) endif                   //rec fr 2.p2
       a1       NbSum(RId) = Partial                        //out of Ack
```

```
                ⇒if RId ∈ Children then NbSum(RId)= Sum(RId).S
                   else NbSum(RId)=0 endif                         //fr 4.p6
                ⇒if RId ∈ Children then NbSum(RId)= (sum s∈Subtree(RId): Sum(s).N)
                   else NbSum(RId)=0 endif                         //SP(RId)
      a2        Npa'out = Npa + RId                                //out of Ack
                ⇒◇ (Npa = Np)                                      //Lemma 1c, Lemma 2, def of Np
5'    p7        case Npa ≠ Np
                ⇒ true                                             //no assertions
5     p7        case Npa = Np                                      //fr 4.a2
      p8           S = (sum s∈Np:NbSum(s)) + N
                   ⇒S = (sum s1∈Others:NbSum(s1)) +
                      (sum s2∈Children:NbSum(s2)) + N              //inv of Np
                   ⇒S = 0 + (sum s2∈Children:NbSum(s2)) + N        //Lemma 1a
                   ⇒S = (sum s2∈Children:
                      (sum s1∈Subtree(s2): Sum(s1).N)) + N         //4.a1
                   ⇒S = (sum s∈{Subtree(s2)| s2∈Children}:
                      Sum(s).N) + N                                //merge 2 sums
                   ⇒S = (sum s∈{i + {Subtree(s2) | s2∈Children}}:
                      Sum(s).N)                                    //N≡Sum(i).N
                   ⇒S = (sum s∈Subtree: Sum(s).N)                  //def of Subtree
6-1   p9           case MyId = Root                                //is Root
7-1   p10             Init(MyId).Ack(MyId, S)
8-1                   SP(i) = true                                 //fr 5.p8&7-1.p10
6-2   p9           case MyId ≠ Root                                //is not Root
      p11             Sum(ParentId).Ack(MyId, Sum(i).S)
7-2                   SP(i) = true                                 //fr 5.p8&6-2.p11
```

end of SP : *Recursive Part*

Lemma 3: Subtree(Root) = Sites.

```
1          Nbs(i) ⊂ Sites                                     //defs of Nbs & Sites
           ⇒ Np(i) ⊂ Sites                                    //def of Np
           ⇒ Children(i) ⊂ Sites                              //def of Children
           ⇒ Subtree(i) ⊂ Sites                               //def of Subtree
           ⇒ Subtree(Root) ⊂ Sites                            //let i=Root
2          assume ∼(Subtree(Root) ⊃ Sites)                    //by assumption
           ⇒ ∃ α∈Sites: ∼(α ⊂ Subtree(Root)) and ParentId(α)=null //subset def
           ⇒ ∀ β∈Subtree(Root): ∼(α ∈Nbs(β))                  //has never called by any site
           ⇒ ∼(α∈Connect(Root))                               //def of Connect
           ⇒ ∼(α∈Sites))                                      //req of Connect
           ⇒ (Subtree(Root) ⊃ Sites)                          //assumption doesn't hold
3          (Subtree(Root) ⊂ Sites) and (Subtree(Root) ⊃ Sites) //fr 1 & 2
           ⇒Subtree(Root)=Sites                               //def of "=" in set
```

sys property: (∀ i∈Sites: SP(i)) and ◇ Init(Root).S= (sum s∈Sites: Sum(s).N)

```
1          Init(Root)                                         //init condition
           ⇒ ∀ i∈Sites: SP(i)                                 //fr SP
           ⇒Sum(Root).S = (sum s∈Subtree(Root): Sum(s).N)
              and Init(Root).Ack(Root, Sum(Root).S)           //let i=Root
           ⇒ Ack(Root, S) and Init(Root).S=Sum(Root).S        //Init(Root) rec Ack from Sum(Root)
           ⇒ Init(Root).S=(sum s∈Subtree(Root): Sum(s).N)     //assignment
           ⇒ Init(Root).S=(sum s∈Sites: Sum(s).N)             //Lemma 3
           ⇒ sys property = true
```

end of verification