Refinement based validation of an algorithm for detecting distributed termination

Mamoun Filali, Philippe Mauran, Gérard Padiou,
Philippe Quéinnec, and Xavier Thirioux

Institut de Recherche en Informatique de Toulouse
118 route de Narbonne, 31062 Toulouse cedex 4, France

Abstract. We present the development and the validation of an algorithm for detecting the termination of diffusing computations. To the best of our knowledge, this is the first one which is based on the maximal paths generated by a diffusing computation. After an informal presentation of the algorithm, we proceed to its rigorous development within the framework of the UNITY formalism and the assistance of the PVS proof system. The correctness of the algorithm is established through a refinement of an abstract model.

1 Introduction

Termination detection of diffusing computations has been extensively studied [13]. Many algorithms [5, 9, 10] have been published to deal with this problem. We propose an algorithm which is based on the observation of final paths of a tree where the nodes represent the local computations, and the edges represent message communication.

The main features of the algorithm are: no assumption about message ordering is made; algorithm messages are piggybacked over those of the computation, and the algorithm greatly reduces the number of control messages. Termination messages are generated once a maximal path of the dynamic tree has been reached. With respect to memory requirements, each node only needs a local counter, the piggybacked data has a fixed size, as well as the collector's data structures. Termination is detected as soon as the collector has received all the maximal path messages.

In order to validate the proposed algorithm, we adopt the UNITY formalism and a refinement based development methodology. The correctness of the algorithm is established through a refinement between a concrete model (associated with the algorithm) expressed as a UNITY program and an abstract model also expressed as a UNITY program. Algorithm properties are derived from those of the abstract model. The development has been mechanized and expressed as theories of the PVS assistant theorem prover [12].

2 Description of the Algorithm

The algorithm observes a diffusing computation according to a specific viewpoint. Instead of reasoning on messages, we interpret communication as a transfer of control. This leads us to introduce path vectors as a new counting mechanism of maximal paths in a tree. After defining a diffusing computation, we describe the actual algorithm.

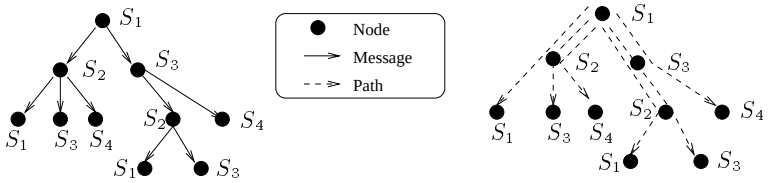

Fig. 1. A message versus path oriented interpretation of a diffusing computation

2.1 Diffusing Computation

We consider a set of processes or nodes $\{P_i \ : 0 \leq i < N\}$ exchanging messages. An initial process P_0 starts the computation by sending a message to a subset of the processes. Then, every process forever repeats the following steps :

`receive(m); handle m; skip, or send up to N messages;` Our model is based on the following assumptions:

- the communication network is connected, asynchronous and reliable;
- processes continuously wait for an input message in passive state, and perform an active step for each available input message;
- each active step is considered to be an atomic action. Such an atomic step ends either "silently" terminating without any applicative message, or by sending one or several messages to other computation processes.

2.2 Termination Detection

We say that a diffusing computation is terminated when all the processes are waiting to receive a message, and no messages are in transit. Due to the atomicity hypothesis, the diffusing computation is terminated when no messages are in transit.

We interpret a diffusing computation as a tree structured spreading of paths. This more abstract viewpoint allows to detect the termination through an algorithm that counts path creations and endings (see Figure 1). More precisely, the algorithm is based the counting of maximal paths starting from the root of the computation tree. A maximal path is defined as a sequence of adjacent edges leading to a leaf. Henceforth, maximal paths starting from the root are called paths.

According to a well-known pattern, a collector process receives the information required to detect the termination.

The algorithm computes the difference $\Delta(i)$ between the number of edges and the number of vertices created by the computation on site i. A process collects these differences, and records the current number T of actual final leaves. By Euler's theorem on planar graphs, the following relation holds at termination (only): $T = \sum_{i=0}^{N} \Delta(i) + 1$

The algorithm evaluates the $\Delta(i)$ terms thanks to the handling of path vectors.

2.3 Path Vectors

Path vectors provide a mechanism to count the number of path creations. Path vectors are vectors of non decreasing integer counters, and their handling is based on the following rules:

– a local counter C_i in each process records the number of locally created paths. These counters are initialized to 0;
– the size of the vector is equal to the number of processes;
– each underlying computation message piggybacks a path vector. A vector of a given path is updated along the spreading of the path. This path vector records the number of new paths issued by the current path, and carries the causal history of these paths.

An initiator process P_{i_0} performs a **Begin** action to assign the first path vector V_0 and the local counter C_{i_0}: $\langle \forall k \neq i_0 \ : \ V_0[k] = 0 \rangle \wedge V_0[i_0] = 1$

Let V be a path vector received by a P_i process. This vector is updated according to the following actions:

– **Split**(p): the computation step ends by sending p messages with $(p > 1)$: new paths are created, and the incoming one follows its way. In this case, the local counter increases by the number of new paths. If p messages are sent, $p - 1$ paths are new. Therefore, the following increment is performed: $C_i := C_i + (p - 1)$. Then, the i'th component of the vector V is assigned the resulting value: $V[i] := C_i$. This updated vector is piggybacked in all sent messages. Every vector leaving a process exactly knows the number of paths created by this process.
– **End**: the computation step ends by sending no message: the path ends. The V vector is sent to the collector. It reports the path ending, and states that a set of paths has been created along its way.

Algorithm of the Collector Thanks to the gathered path vectors, the collector progressively evaluates the number of created and ended paths. It handles a path vector MT and a counter T. In the initial state, they are equal to zero. The collector gathers the path vectors until it can decide the termination.

```
process Collector
   integer MT[N] = [0, ... , 0];     /* maximal vector */
   integer T = 0;                    /* counter of terminated paths */
 repeat
   integer V[N];
   receive(V);
   MT, T := max(MT, V), T + 1;
   Term := (∑ MT = T);
 until (Term) /* Detect */
```

The timing diagram of Fig. 2 illustrates the computation behavior.

3 The UNITY Formalism

The UNITY formalism consists of two parts: a programming language, based on transition systems, and a specification language, based on a linear temporal logic [8], to express behavioral properties of these transition systems. A comprehensive presentation of the UNITY formalism can be found in [3] or [11].

As UNITY is a modeling language, it accepts any notation, or data type, with a clearly defined semantics. In this paper, we will mostly use:

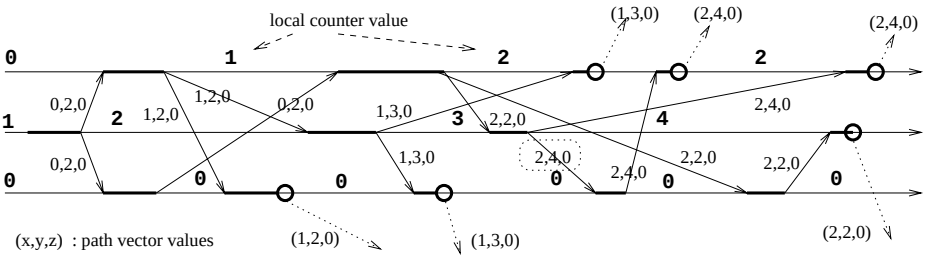

Fig. 2. A computation example

– the `bag` type, i.e. multisets. The operators on `bag`s are defined by overloading the usual set operators ($\cup, \emptyset, \epsilon$[1], ...), $k * e$, where k is a natural, is the multiset consisting of k occurrences of e.
– the `Vector` type, which corresponds to integer vectors, indexed by process identifiers. `V with [(i):=e]` denotes a vector that is identical to V, except for the ith component, where the value is e.

3.1 UNITY Logic Predicates

Reasoning about UNITY programs generally involves two kinds of predicates:

– properties whose definition refers to the program text. UNITY's original (inductive) temporal operators are defined in this way [3].
– properties which are defined from the set of execution sequences that result from the execution of a UNITY [14, 15, 11] program. An execution sequence σ resulting from a run of a program F verifies the following properties:
  - valid initial state: $F.init(\sigma(0))$
  - transition by statement s: $\langle \forall i :: \langle \exists s : s \in F.assign :: \sigma(i+1) = s(\sigma(i)) \rangle \rangle$
  - equity: $\langle \forall s : s \in F.assign :: \langle \exists_i^\infty :: \sigma(i+1) = s(\sigma(i)) \rangle \rangle$

  Let F be a program, and $\mathcal{S}(F)$ the set of such sequences, we define:

$$\textbf{stable}\ p\ \textbf{in}\ F \equiv \langle \forall \sigma, i : \sigma \in \mathcal{S}(F) :: p(\sigma(i)) \Rightarrow p(\sigma(i+1)) \rangle$$
If p is true, it remains true.

$$\textbf{alwt}\ p\ \textbf{in}\ F \equiv \langle \forall \sigma, i : \sigma \in \mathcal{S}(F) :: p(\sigma(i)) \rangle$$
p is always true.

$$p \mapsto q\ \textbf{in}\ F \equiv \langle \forall \sigma, i : \sigma \in \mathcal{S}(F) :: p(\sigma(i)) \Rightarrow \langle \exists j : j \geq i :: q(\sigma(j)) \rangle \rangle$$
(leadsto) If p is true, q will become true.

3.2 Refinements

Several definitions of refinement have been proposed in the field of reactive programs [1, 7, 2]. In this paper, our approach is similar to Abadi and Lamport's [1], as their formal framework is very close to the UNITY formalism; We mostly used the results about

[1] ϵ denotes the choice of a *given* element in a non-empty set.

the preserving of safety properties when an "abstract" model is refined by a "concrete model". Informally, [1] defines a refinement as a mapping φ from states of the concrete model, to states of the abstract model such that:

- φ takes initial states of the concrete model into initial states of the abstract model.
- φ maps state transitions in the concrete model into state transitions in the abstract model, i.e.: if (c, c') is a pair of states of the concrete model that corresponds to a transition, then $(\varphi(c), \varphi(c'))$ corresponds to a transition of the abstract model.

4 Validation

In this section, we study a model for the detection termination algorithm. After presenting the basic properties to be satisfied by a detection algorithm, we present the validation structure within the UNITY framework.

4.1 Specification of the Termination

In the following, we take the *bag* data type as the basic data structure for modeling the communication medium. To each process `p`, we assign a bag `B[p]`. Inserting an element into `B[p]` corresponds to sending a message to `p`, extracting an element from `B[p]` corresponds to the reception of message by `p`. The elements of `B[p]` represent the messages that have been sent to `p` but not yet received by `p`. Thanks to the atomicity hypothesis (§2.2), we can specify termination as follows:

- safety: there is no false detection: **alwt** Term $\Rightarrow \langle \forall p : B[p] = \emptyset \rangle \quad (\text{term}-0)$
- liveness: a termination is eventually detected: $\langle \forall p : B[p] = \emptyset \rangle \mapsto \text{Term} \quad (\text{term}-1)$

In the following development, we are mainly concerned by safety. A sketch of the liveness proof is given in Sect. 4.7.

4.2 Structure of the Validation

During the validation process, we have been concerned by two aspects:

- first, defining a sound model of the underlying communication network as well as of the underlying diffusing computation. We have chosen the bag data structure and the UNITY superposition mechanism.
- secondly, stating the basic idea of the algorithm in a simple way. We have elaborated an abstract model, whose invariant allows to conclude when termination has occurred.

Figure 3 illustrates the structure of the validation, where: `dc` is a model for a diffusing computation, `t` is an abstraction of the basic idea of the algorithm, and `Term_Max` is a model of the termination algorithm. The program `aux_Term_Max` introduces an auxiliary variable and corresponding statements updating it. This program is used for the proof.

The validation consists in showing that the concrete model, namely the program `Term_Max` is an instance of a diffusing computation, and then, satisfies the termination properties. The termination properties will be established by "inheriting" those of the abstract and auxiliary programs.

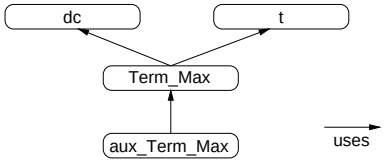

Fig. 3. Structure of the validation

4.3 Diffusing Computation Pattern

In the following, we give the pattern of a diffusing computation as an incomplete UNITY program. It belongs to the user to fill in the pattern to obtain a program[2]. In this pattern, message type (<M>) must be instantiated, new variables can be declared (<v>) and initialized (<iv>). It is also possible to superpose new statements either synchronously (<is>) or asynchronously (<ia>) .

```
Program dc
declare      B : array Process of bag of <M>
             <v>
initially    B = [<d>,∅, ... ]
             <iv> --  superposed variables initialization
assign --  Split
  ⟨[] p,D : D ≠ ∅ ∧ B[p] ≠ ∅ :
      B[p]:= B[p] − ϵ(B[p]) if p ∉ D
          ∼  B[p] − ϵ(B[p]) + <m> if p ∈ D
      ‖ ⟨‖ f : f ∈ D − p : B[f] := B[f] + <m> ⟩
      ‖ <is>        --  synchronous superposition
  ⟩
[] ⟨[] p : B[p] ≠ ∅ : B[p]:= B[p] − ϵ(B[p]) ‖ <is> ⟩ --  End
[] <ia>     --  asynchronous superposition
end dc
```

This pattern is a model for the program in Sect. 2.1:

- the first statement (commented *Split*) is a model for the atomic sequence:
  `receive(m); handle(m); send up to N Messages`
  where p is the sending process, and D is the set of destination processes[3]: receiving a message consists in extracting a given message ($\epsilon(B[p])$) from the input box of p; sending a new message to a process f consists in adding it to its input box.
- the second statement (commented *End*) is a model for the atomic sequence:
  `receive(m); handle(m); skip`

4.4 The Concrete Model

The concrete model is obtained by instantiating the diffusing computation pattern. Since the properties of the concrete model are established from those of the abstract model and the auxiliary model, we just give here the UNITY expression of the concrete model.

[2] Such a pattern could be implemented by a syntax editor.

[3] Two cases must be distinguished, since a process can send a message to itself.

```
Program Term_Max
declare  B : array Process of bag of Vector
--  variables superposition
  C,MT : Vector;          BT : bag of Vector
  T: nat                  Term: bool
always  nc(p,D) = C[p]+|D| −1                --   new counter
        nv(v,p,D) = v with [(p):= nc(p,D)] --   new vector
initially  ⟨∃ i, V :: (∀ k: k ≠ i ⇒ V[k] = 0 ∧ C[k] = 0) ∧
                      V[i] = 1 ∧ C[i] = 1 ∧ B = {V} ⟩
--  superposed variables initialization
  MT,BT,T,Term= [0, ... ],∅,0,false
assign  --  Split
  ⟨[] p,D : D ≠ ∅ ∧ B[p] ≠ ∅ :
       B[p]:= B[p] − ϵ(B[p]) if p ∉ D
           ∼  B[p] − ϵ(B[p]) + nv(ϵ(B[p]),p,D) if p ∈ D
     || ⟨||f: f ∈ D − p: B[f]:=B[f] + nv(ϵ(B[p]),p,D) ⟩
     || C[p]:= nc(p,D) ⟩--  synchronous superposition
[] ⟨[] p: B[p]≠∅:B[p]:= B[p]−ϵ(B[p]) || BT:=BT+ϵ(B[p])⟩ --  End
--  asynchronous superposition
[] BT,T,MT:= BT−ϵ(BT),T+1,max(MT,ϵ(BT)) if BT ≠ ∅   --  Term
[] Term := Σ MT = T if T > 0    --  Detect
end Term_Max
```

In this UNITY program:

- The **always** section should be understood as a declaration of macros.
- The statements commented by *Split* and *End* are the actions of a computation process. The statements commented by *Term* and *Detect* are the actions of the collector process.
- The bags `B[_]` and `BT` represent the communication medium. `MT,T,Term` are the variables of the `Collector` process, each element of the vector `C` represents the local counter which is updated by computation processes (Sect. 2.3).
- `nv` is the new vector sent, and `nc` is the new value of the local counter.

In order to have a program structure similar to the model described in terms of processes of the first part of the paper, we could have expressed this program as follows:

$$\text{Term\_Max} = \langle [] p :: \text{Computation}[p] \rangle [] \text{Collector}$$

We have chosen the monolithic structure to stay within a classic framework for refinements and to reason in a simple way with respect to liveness properties.

4.5 The Abstract Model

The abstract model expresses the basic mechanism of the algorithm which consists in counting the elements of the bag `g` through a distributed counter : the vector `C`.

Program t
declare c: Vector g: bag **of** Vector
always nc(p,d) = c[p] + d − 1
nv(v,p,d) = v with [(p):= nc(p,d)]
invariant
⟨∀A::(A ≠ ∅ ∧ A ⊂ g ∧ Σ max(A) ≤ |A|) ⇒ A = g⟩-- *maximal*
∧ g ≠ ∅ ∧ c = max(g) ∧ |g| = Σ max(g) -- *card_max*
initially |g| = 1 ∧ Σ max(g) = 1
assign ⟨[]v,p,d: v ∈ g ∧ 0<d: c[p],g:= nc(c,p),g−v ∪ d∗nv(v,p,d)⟩
end t

In the abstract model[4], we have represented the leaves of the tree by the bag g. In the full report [6], we give the correctness proof for the invariant.

4.6 The Auxiliary Model

The auxiliary model enriches the concrete model with an auxiliary variable `a : bag of Vector` which represents the bag of the messages which have been received and handled by the collector. This variable is initialized to the empty bag, and is updated each time a message is delivered to the collector. The following property holds in the auxiliary program (its Unity program text is in the full report [6]):

invariant
⟨∀ A::(A ≠ ∅ ∧ A ⊂ g ∧ ∑max(A) ≤ |A|) ⇒ A = g⟩
∧ g ≠ ∅ ∧ C = max(g) ∧ |g| = ∑max(g)
∧ MT = max(a) ∧ T = |a|
∧ a = ∅ ⇒ ¬Term
∧ Term ⇒ ∑MT = T

The properties of the auxiliary model are inherited from those of the abstract model. For this purpose, we have shown that the auxiliary model is a refinement of the abstract model through the functional relation:

$$(g = \cup_p B[p] \cup BT \cup a) \wedge \text{aux\_term\_max}.C = t.c \qquad (1)$$

The abstract bag g is the union of B[p] the processes bags, BT the bag containing the messages sent to the collector but not yet accepted, and a the bag of the messages sent to the collector and accepted.

In order to show the refinement, bag insertions and extractions of the auxiliary model have to be mapped to operations on the abstract variable g such that (1) is preserved. Thus, we map the statement Split to the unique explicit statement of the abstract model, and we map the statements End, Term and Detect to the implicit statement Skip. The main property established on the auxiliary program is:

$$\textbf{alwt}\ \text{Term} \Rightarrow \langle \forall p : B[p] = \emptyset \rangle\ \textbf{in}\ \texttt{aux\_term\_max}$$

In a similar way, the concrete model can inherit the properties of the auxiliary model since the concrete model refines the auxiliary model. It follows, that the safety of the detection (term-0) is satisfied by the concrete model.

[4] Following the path initiated by [4], we put together a Unity program and its properties.

4.7 Liveness

Informally, liveness relies on the following arguments:

- once the bags of the computation processes are empty, they remain empty;
- when the bags of the computation processes are empty, the bag of the collector eventually decreases;
- once the bags of the computation processes are empty and the bag of the collector is empty, termination is eventually detected.

It follows that the liveness proof relies mainly on the transitivity of $\mapsto$ and on the fact that finite bag inclusion is well-founded. The following lemmas, combined with the **alwt** properties of the algorithm, establish the liveness of the termination (term-1).

stable $\langle \forall p : B[p] = \emptyset \rangle$
$(\langle \forall p : B[p] = \emptyset \rangle \wedge BT = \mathcal{B} \wedge BT \neq \emptyset) \mapsto (\langle \forall p : B[p] = \emptyset \rangle \wedge BT \subsetneq \mathcal{B})$
$(\langle \forall p : B[p] = \emptyset \rangle \wedge BT = \emptyset) \mapsto$ Term

4.8 Mechanizing the Development

We have encoded the semantics of the preceding models in the typed logic of PVS [12]. The typed logic allowed us to encode the generic aspects of patterns as well as the parameterized aspects of the problem. The main theories of this development and their relationships are given in Fig. 4. `programs` and `refinements` are generic theories and contain results on UNITY logic. The PVS development can be found in [6].

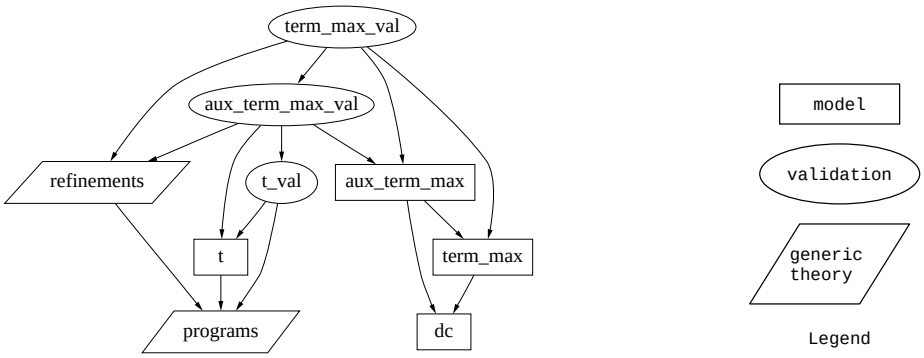

Fig. 4. Theories of the algorithm's validation

The expression of UNITY programs in the PVS system is straightforward: the syntax of the PVS specification language is close to the imperative description of models.

5 Conclusion

This paper presents an original case study, in that it shows the use and the integration of formal models in the design process of a new algorithm. The other specificity of this case study lies in its area of application : distributed algorithms and systems.

As regards the first point, the UNITY formalism provided a sound grounding for the specification and design of the algorithm, prior to its validation. The design followed a refinement based approach: a first (abstract) model describes the main idea of the algorithm, and a second (concrete) model refines and maps the algorithm to a distributed environment. The formal validation was carried out with the PVS assistant theorem prover [12]. PVS turned out to be an appropriate tool, w.r.t. the ability to easily express UNITY programs, properties and their development. The validation provided key elements for a deeper understanding of the algorithm, and for improving the model and the informal proof.

As regards the second point, the modeling reflects the hypotheses on the communication network by using multisets, instead of sequences, to represent pending messages.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. Technical Report 29, DEC, August 1988.
2. J.R. Abrial. *The B-Book Assigning programs to meanings*. Cambridge University Press, 1996.
3. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
4. Michel Charpentier. *Assistance à la Répartition de Systèmes Réactifs*. Thèse de doctorat, Institut National Polytechnique de Toulouse, France, November 1997.
5. E.W.D. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(4):1–4, 1980.
6. Mamoun Filali, Philippe Mauran, Gérard Padiou, Philippe Quéinnec, and Xavier Thirioux. Un algorithme de terminaison de calcul diffusant par évaluation des chemins maximaux. Rapport de recherche 99-22-R, Institut de Recherche en Informatique de Toulouse, November 1999. 35 pages.
7. N. Lynch and F. Vaandrager. Forward and backward simulations – part I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.
8. Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: specification*. Springer-Verlag, 1992.
9. Friedemann Mattern. Algorithms for distributed termination detection. *Distributed Computing*, 2(3):161–175, 1987.
10. Friedemann Mattern. Virtual time and global state in distributed systems. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.
11. Jayadev Misra. A logic for concurrent programming. Technical report, The University of Texas at Austin, April 1994.
12. Sam Owre, John M. Rushby, and Natarajan Shankar. PVS: A prototype verification system. In *11th Int'l Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer-Verlag, June 1992.
13. Michel Raynal. *Synchronisation et État Global dans les Systèmes Répartis*. Collection Direction Etudes-Recherches EDF. Edition Eyrolles, 1992.
14. Beverly A. Sanders. Eliminating the substitution axiom from UNITY logic. *Formal Aspects of Computing*, 3(2):189–205, April–June 1991.
15. Rob T. Udink and Joost N. Kok. On the relation between Unity properties and sequence of states. Technical Report RUU-CS-93-07, Utrecht University, Dept. of Computer Science, February 1993.