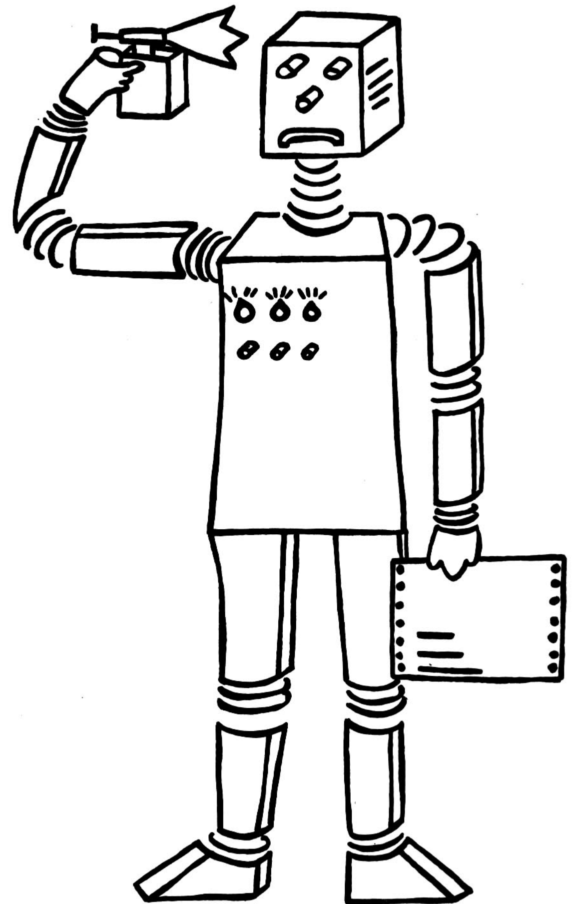
SEKI-Working Paper

Fachbereich Informatik
Universität Kaiserslautern
Postfach 3049
D-6750 Kaiserslautern 1, W. Germany



TENUA

A Test Environment for Unification Algorithms

R. Scheidhauer und G. Seul
Universität Kaiserslautern, FB Informatik - AG Siekmann
Postfach 3049, D-6750 Kaiserslautern, FR Germany

TENUA

A Test Environment for Unification Algorithms

R. Scheidhauer und G. Seul
Universität Kaiserslautern, FB Informatik - AG Siekmann
Postfach 3049, D-6750 Kaiserslautern, FR Germany

Contents

0. Introduction

1. Facilities for the Implementation of Unification Algorithms
   1.1 Basic Definitions
   1.2 Representation of Terms and Substitutions
   1.3 Term Conversion
   1.4 Input and Output of Terms and Substitutions
   1.5 Arity-Check
   1.6 Interface for Unification Functions

2. Description of the Implemented Unification Algorithms
   2.1 The Robinson Algorithm
   2.2 The Martelli/Montanari Algorithm
   2.3 The Escalada/Ghallab Algorithm

3. Description of the Implemented Term Generators
   3.1 Parametrized Term Generators
   3.2 Generators for Random Terms
      3.2.1 GENTERM-RND1
      3.2.2 GENTERM-RND2

4. Facilities for Testing Correctness and Efficiency of Unification Algorithms
   4.1 Facilities for Testing Correctness
   4.2 Facilities for Testing Efficiency
      4.2.1 Efficiency Test Using Standard Terms
      4.2.2 Efficiency Test Using Random Terms

5. Conclusion

6. References

7. Appendix

0. Introduction

In this paper we describe TENUA, a Test ENvironment for Unification Algorithms for first order terms.
In its essence the unification problem in first order logic can be expressed as follows: Given two terms containing some variables, find, if it exists, the simplest substitution (assignment of some term to every variable) which makes the two terms equal. Since Herbrand's original work (Herbrand 1930), unification has been the subjekt of several research works, mainly settled in the field of artificial intelligence. The first unification algorithm, introduced by Robinson 1965, constituted the central step of the resolution principle, which is frequently used in theorem proving and logic programming like PROLOG. Resolution, however, is not the only application of the unification algorithm. In fact its pattern matching nature often can be exploited in cases where symbolic expressions are dealt with, for instance type checkers for programming languages with a complex type structure, rewriting systems and some knowledge representation formalisms in AI. Because in all these applications unification constitutes the elementary operation, its performance effects in a crucial way their global efficiency. Since the Robinson algorithm has a exponential worst case complexity, soon linear (Paterson/Wegman 1978, Escalada/Ghallab 1987) or almost linear algorithms (Martelli/Montanari 1983) were developed.
The choice of the appropriate unification algorithm for some application is facilated by TENUA, a tool which allows comfortable implementation and analysis of unification algorithms. It provides the user with:

- Facilities for the implementation of unification algorithms such as an interface for input and output of terms and substitutions (including an arity check), and functions for term conversion (see chapter 1).
- Implemented unification algorithms (Robinson 1965, Martelli/Montanari 1982, Escalada/Ghallab 1987) giving a practical measure of efficiency (see chapter 2).
- Facilities for the comparision of unification algorithms such as statistical functions (see chapter 4) and parameterized generators for "standard" and "random" terms (see chapter 3). This allows the user to produce term pairs appropriate to his application and so to test the efficiency of unification algorithms on "real" conditions (see example 4.2).

TENUA is implemented in COMMON LISP on Apollo Domain Workstations. Except the online documentation (HELP-facility) it is machine independent and can be loaded in any COMMON LISP environment.

1. Facilities for the Implementation of Unification Algorithms

1.1 Basic Definitions

To describe the unification problem for first order terms we briefly introduce basic definitions and results:

The set T of *terms* over a countable set V of variables and a countable set F of function symbols (including constants) is recursively defined by:

$$t \in T \quad \text{iff} \quad t \in V \text{ or } \exists f \in F, t_1,\dots,t_n \in T \text{ with } t = f(t_1,\dots,t_n)$$

A *substitution* is a mapping γ: V → T for which $\gamma(x) = x$ except on a finite part of V. A substitution can therefore be described by a finite set $\{x_i \leftarrow t_i;\ 1 \le i \le n\}$. Substitutions are naturally extended as homomorphisms T → T by $\gamma f(t_1,\dots,t_n) := f(\gamma t_1,\dots,\gamma t_n)$.

A *unifier* σ of two terms s and t is a substitution σ with $\sigma(t) = \sigma(s)$. It is called a *most general unifier (mgu)* of these terms, iff for every unifier μ of t and s there is a substitution δ with $\mu = \delta \circ \sigma$. There always exists a mgu for two unifiable terms.

The *unification problem* $\langle s = t \rangle$ is the problem, to find such an mgu for two terms s and t.

1.2 Representation of Terms and Substitutions

TENUA offers two different representations of terms: the *string representation*, which allows a mathematical notation and therefore is very appropriate for input and output (but unsuitable for term manipulation) and the *representation as LISP S-expressions*, which is very handy for algorithmic applications (have always in mind that LISP is our programming language). Now we give both notations in more detail. Functions for conversion, input, output and arity check of terms (in the above representation) are described in chapter 1.3 - 1.5.

Terms as Strings

This representation corresponds to the usual mathematical notation, including the following convention: Variables are represented by letters u, v, w, x, y, z, constants by a, b, c, d, e and function symbols by f, g,..., s or t. To expand the number of symbols it is allowed to add a number to a letter.

Example: "a"; "x5"; "f(a, b, g(x, u))"; "h4711(f12(a3), x2, z)" are legal terms.

Terms as LISP S-expressions (list representation)

Here variables, constants and function symbols are represented as LISP atoms with the same restrictions as for string terms. Composed terms are LISP lists, where the head corresponds to

the functor and the tail to the argument list (already LISP S-expressions).
Example: A; X5; (F A B (G X U)); (H4711 (F12 A3) X2 Y).

For the same reason as for terms TENUA accepts two different representations for substitutions.

Substitutions as Strings
This representation is very similar to mathematical notation and therefore suitable for input and output. It always begins with a "{" and ends with a "}", enclosing a finite number of pairs (of the form) <variable> <-- <term> seperated by commas.
Example: "{x ← a, y ← f(z, g(b))}" or "{}" (empty substitution).

Substitutions as Association Lists
To work more easily with substitutions in LISP they are also represented as association lists. The elements of such an association list are of the form (variable . term) . The substitutions of the previous example now are represented as ((X . A) (Y. (F Z (G B)))) and NIL for the empty substitution. Notice however, that by LISP convention the first substitution is printed as: ((X . A) (Y F Z (G B))).

See also chapter 1.4 for an output function for substitutions.

1.3 Term Conversion

TENUA supplies the following two functions to transform the string and the list representation of terms into each other:

STRING-TO-TERM gets a term t in string representation and returns the list representation of t. Supernumerary closing parantheses and blanks are ignored.
Example: (STRING-TO-TERM "f(a, g(y))))))" has value (F A (G Y)).

TERM-TO-STRING gets a term t in list representation and returns the string representation of t.
Example: (TERM-TO-STRING '(F (G X) Y Z)) returns "f(g(x), y, z)".

1.4 Input and Output of Terms and Substitutions

To take the implementation of input/output procedures from the user, TENUA provides some comfortable built in functions. As already explained in chapter 1.2 list representation of substitutions and terms constitutes the working data format. The following functions describe

the interface for input/output of working data:

The function TERM-IN reads new lines from terminal as long as the number of closed parantheses is less than the number of open parantheses. Blanks and supernumerary closing parantheses are ignored. The input string is transformed into the corresponding list representation. Optionally the user can specify the input stream, from which TERM-IN reads. Example: (TERM-IN stream1) reads a term from stream1 and returns its list representation.

TERM-OUT performs the inverse operation. It gets a term in list notation and prints its string representation to the standard output. As optional arguments the user can specify the file on which TERM-OUT writes and the minimum of symbols per line (see online documentation).

Example: (TERM-OUT '(F A (G Y))) prints "f(a, g(y))" to standard output ;
(TERM-OUT '(F A (G Y)) "testfile") prints "f(a, g(y))" to the file "testfile".

The function SUBST-OUT gets the result of a unification problem in working format, that is, either an association list (see chapter 1.2) or one of the both LISP atoms CLASH or CYCLE, and writes its string representation cleverly arranged on standard output. Again the user can optionally specify the name of the output file and the minimum length per line.

Example: (SUBST-OUT '((X . A) (Y . (F B)))) writes "{ x ← a, y ← f(b)}" to standard output.

1.5 Arity Check

The user has the possibility to check a unification problem on consistence, that is to reject term pairs in which one function symbol occurs with different arities:
CHECK-ARITY gets two terms s and t in list representation and returns T(rue) if each function symbol in s and t is used with consistent arity, otherwise the pair of subterms in which the difference occured.
Example:
(CHECK-ARITY '(F (G A)) '(F (G B))) returns T
(CHECK-ARITY '(F (G A)) '(F (G B C))) returns ((G A) (G B C)).

1.6 Interface for Unification Functions

If the user wants to implement a unification algorithm in LISP without wasting time by such problems as cosmetic preparation of output data, he can use TENUA's UNIFY. His unification

function merely must satisfy the following input/output specification: it gets two terms in list representation and returns a substitution in explicit form represented as an association list or one of the two atoms CLASH or CYCLE. UNIFY gets the name xyz of the users unification function and two terms s and t either in string or in list representation. The arity of the function symbols of s and t is checked and both terms are transformed into list representation (if necessary). Subsequently xyz is applied to them. The result is transformed into string representation and displayed on screen.

Built in unification functions are:
ROB (Robinson), MM (Martelli/Montanari) and EG (Escalda/Ghallab) (see chapter 2)
Correct calls of unify are for example:

```
    (UNIFY 'ROB '(F A) '(F X) )
        (has the effect that "{x ← a}" is printed on screen)
or  (UNIFY 'XYZ "f(a)" "f(x)").
```

2. Description of the implemented Unification Algorithms

2.1 The Robinson Algorithm

Robinson's algorithm, developed in 1965, was the first known unification algorithm for first order terms.
It is based on the following idea :
Given a unification problem $\langle s = t \rangle$, two cases are distinguished:

1) If one of the two terms is a variable x, the so called *Occurcheck* is performed, which means it is checked, if x occurs in the other term. If the test is positive we have a *cycle* and $\langle s = t \rangle$ is not unifiable, otherwise the substitution $\{x \leftarrow t\}$ is a mgu of $\langle x = t \rangle$.

2) The two terms are of the form $s = f(s_1, s_2,\ldots,s_n)$ and $t = g(t_1, t_2,\ldots, t_m)$, where constants are considered as 0-ary functions. If the functors f and g are different we have a *clash* and s , t are not unifiable. Otherwise the Robinson algorithm is sequentially applied to the corresponding argument pairs ($\langle s_1 = t_1 \rangle$, $\langle s_2 = t_2 \rangle$...), where the mgu's of preceding pairs are applied to the subsequent pairs before their recursive treatment. If a cycle or clash is produced in the recursion, s and t are not unifiable, otherwise the mgu is the composition of the mgu's of the argument pairs.

This can be represented in the following algorithmic notation:

```
Algorithm ROBINSON-UNIFY
Input:  A pair of terms ⟨ s, t ⟩
Output: if ⟨ s, t ⟩ are unifiable, a mgu σ of ⟨ s = t ⟩, CLASH or CYCLE otherwise
BEGIN
IF  one of the two terms s and t is a variable x
THEN  let u be the other;
         IF  x=u
         THEN  σ:={ }
         ELSIF  Occur(x,u)
         THEN  exit (CYCLE)
         ELSE  σ:={x ← u}
         FI
ELSE  let s=f(s1, s2,..., sn) and t=g(t1, t2,..., tm);
        IF  f≠g
        THEN  exit (CLASH)
        ELSE  σ:={ };
                FOR k:=1 TO n DO
                  τ:=ROBINSON-UNIFY(σ(sk), σ(t)) ;
                  σ:= τ σ
                ENDFOR
        FI
FI
END
```

Example:

$U = \langle f(g(x), x), f(g(y), a) \rangle$

$\sigma := \{\}$

$U_1 = \langle \sigma(g(x)), \sigma(g(y)) \rangle = \langle g(x), g(y) \rangle$

$\tau_1 = \sigma_{1:} = \{\}$

$U_{11} = \langle \sigma_1(x), \sigma_1(y) \rangle = \langle x, y \rangle$

$\tau_{11} = \{x \leftarrow y\}$

$\tau_1 = \sigma_1 = \tau_{11}\, \sigma_1 = \{x \leftarrow y\}\ \{\} = \{x \leftarrow y\}$

$\sigma := \tau_1\, \sigma = \{x \leftarrow y\}$

$U_2 = \langle \sigma(x), \sigma(a) \rangle = \langle y, a \rangle$

$\tau_2 = \{y \leftarrow a\}$

$\sigma := \tau_2\, \sigma = \{y \leftarrow a\}\{x \leftarrow y\}$

$= \{y \leftarrow a, x \leftarrow a\}$

We have implemented Robinson's algorithm with terms represented as lists,which are a kind of

tree structure. Therefore in worst case it has exponential time and space complexity (depending on the term size), caused by the excessively increasing term copies in substitution application at the recursive call. The classical example is:
$U = \langle f(x_1, x_2, ..., x_n), f(g(x_0,x_0), g(x_1,x_1), \ldots, g(x_{n-1}, x_{n-1})) \rangle$ (see also chapter 3).
In TENUA the Robinson Algorithm can be called using the UNIFY function (see chapter 1.5).
Example: The evaluation of the call (UNIFY 'ROB "g(f(x),x)" "g(f(y),a)") returns
$\{x \leftarrow a, y \leftarrow a\}$.

2.2 The Martelli/Montanari Algorithm

The algorithm of Martelli/Montanari was developed in 1982. However it is considered to be one of the most efficient unification algorithms. First of all we give some basic definitions:
A *multiset of consistent terms* (no direct functor clash) will be represented as a *multiterm*.. A multiterm can either be empty or of the form $f(\langle S_1,M_1\rangle, \ldots, \langle S_n,M_n\rangle)$, where the S_i are sets of variables, the M_i are multiterms and S_i and M_i cannot both be empty.
Example: The multiset of consistent terms $\{f(x,g(a,y)),f(b,x),f(x,y)\}$ is represented by the multiterm $f(\langle\{x\},b\rangle,\langle\{x,y\},g(\langle\emptyset,a\rangle,\langle\{y\}, \emptyset \rangle)\rangle)$.
A *multiequation* is of the form S=M, where S is a nonempty set of variables and M is a multiterm. The following algorithm merges two multiterms M' and M" into one, if they are consistent, otherwise it fails.

```
merge( M', M") =
  CASE M' OF
    Ø : M",
    f'(⟨S'1,M'1⟩,...,⟨S'n,M'n⟩):  let M"= f'(⟨S'1,M'1⟩,...,⟨S'n,M'n⟩);
                                  IF f'=f" AND merge(M'i,M"i)≠failure
                                  THEN f'(⟨ S'1 ∪ S"1, merge(M'1, M"1)⟩,...,
                                         ⟨ S'n ∪ S"n, merge(M'n, M"n)⟩)
                                  ELSE failure
                                  FI
```

The *common part* of a multiterm M is also a multiterm and defined as follows:

```
COMMONPART( f(⟨S1,M1⟩,...,⟨Sn,Mn⟩)) = f(P1,...,Pn)
      where Pi = IF Si = Ø
                 THEN COMMONPART(Mi)
                 ELSE ANYOF(Si)
                 FI
```

and the function $ANYOF(S_i)$ returns an element of the set S_i.

The counterpart to the common part of a multiterm is its *frontier*, a set of multiequations.

$$\text{FRONTIER}\ (\ f(\langle S_1,M_1\rangle,\ldots,\langle S_n,M_n\rangle)) = F_1 \cup \ldots \cup F_n$$

$$\text{where } F_i = \text{ IF } S_i = \emptyset \text{ THEN FRONTIER}(M_i) \text{ ELSE } \{S_i{=}M_i\}\quad (i=1,\ldots,n)$$

Given two terms s and t the idea of Martelli/Montanari is to distribute the variables of s and t over equivalence classes of terms, corresponding to the variable bindings of the mgu. The equivalence classes are represented as multiequations. We always consider a system of multiequations R=(T,U) with an unsolved part U and a solved or triangular part T.

By every step of the unification algorithm a multiequation is transfered from the U-part to the T-part, until the U-part is empty.

The initial system has an empty T-part and its U-part contains the following multiequations: the first is of the form $\{x_0\}=\{s,t\}$ (with a new variable x_0); all other multiequations in U are of the form $\{y\}=\emptyset$, where y is a variable occuring in s or t. Vice versa every variable in s or t corresponds to such a multiequation in U. U therefore contains n+1 multiequations, with n = number of variables in s and t. For reasons of efficiency (in multiequation selection) every multiequation S=M of U comprises a counter for the sum of occurrences of the variables of S in the right hand sides of U.

<u>Example</u>: The initial system belonging to s = f(x,g(y,z),y,b), t = f(g(h(a,v),y),x,h(a,u),u) is :

U: {[0] {x}=f(⟨{x},g(⟨∅,h(⟨∅,a⟩,⟨{v},∅⟩) ⟩,⟨{y},∅⟩) ⟩,
⟨{x},g(⟨{y},∅⟩,⟨{z},∅⟩) ⟩,
⟨{y},h(⟨∅,a⟩,⟨{u},∅⟩) ⟩,
⟨{u},b⟩)
[2] {x}=∅
[3] {y}=∅
[1] {z}=∅
[2] {u}=∅
[1] {v}=∅}
T: {}.

While the U-part is nonempty we search for a multiequation in U with counter 0. If no such multiequation exists, every variable of U must occur in the right hand side and therefore in the substituting term of at least one other variable in U (stop with cycle!).

Otherwise that multiequation is removed from the U-part, its common part is added to the T-part, the multiequations of its frontier are merged with the variable corresponding multiequation of U (compactification: here clashes are found!) and the counters are adjusted.

When the U-part is empty the solution can be obtained in explicite form by substituting backward the left hand side variables of T.

Algorithmic notation:

REPEAT

1) Select a multiequation S=M of U such that the variables in S do not occur elsewhere in U (counter=0). If a multiequation with this property does not exist stop with failure(cycle).
2) IF M is empty

    THEN transfer S=M from U to the end of T

    ELSE 1) compute the commonpart C and the frontier F of M

    2) Transfer S=C from U to the end of T

    3) Merge the multiequations of F with the corresponding multiequations of U and adjust the counters (if merge fails stop with clash).

    FI

UNTIL U is empty.

Example: For reasons of readability we will not use multiterm notation, which indeed may be appropriate for computers but not for human.

s and t see previous example:

initial system:

U_0: { [0] {x_0}={f(x,g(y,z),y,b),f(g(h(a,v),y),x,h(a,u),u)},
[2] {x}= Ø
[3] {y}= Ø
[1] {z}= Ø
[2] {u}= Ø
[1] {v}= Ø}

T_0: {}.

U_1: { [0] {x}={g(h(a,v),y),g(y,z)},
[2] {y}= {h(a,u)}
[1] {z}= Ø
[1] {u}= {b}
[1] {v}= Ø}

T_1: { {x_0}=f(x,x,y,u)}.

U_2: { [0] {y,z}={h(a,u),h(a,v)},
[1] {u}= {b}
[1] {v}= Ø}

T_2: { {x_0}=f(x,x,y,u),
{x} =g(y,z)}.

U_3: { [0] {u,v}={b}}
T_3: { $\{x_0\}$=f(x,x,y,u),
{x} =g(y,z),
{y,z}=h(a,u)}.

U_4: Ø
T_4: { $\{x_0\}$=f(x,x,y,u),
{x} =g(y,z),
{y,z}=h(a,u)
{u,v}=b}.

Solution in explicit form by backward substitution:
mgu(s,t)={u ← b , v ← b , y ← h(a,b) , z ← h(a,b) , x ← g(h(a,b),h(a,b))}.
The Martelli/Montanari algorithm has complexity O(n log(n)) provided there is a direct (and not a sequential) access to a variable's equivalence class (using the UNION-FIND algorithm even a quasilinear complexity is reached). In our implementation we guaranteed that property by employing hash tables to represent the actual variable bindings. This causes a relative high amount in administration for small examples, but that is a general draw back of Martelli/Montanari's algorithm.
In TENUA you can call the Martelli/Montanari's algorithm by using the UNIFY function (see chapter 1.5).
Example: (UNIFY 'MM "g(f(x),x)" "g(f(y),a)") returns the unifier in *sequential* form
while (UNIFY 'MM-MULTAUS "g(f(x),x)" "g(f(y),a)") returns the unifier in *explicite* form.

2.3 The Escalada/Ghallab Algorithm

The algorithm of Escalada/Ghallab is the most recent unification algorithm that TENUA provides. It was published in January 1987 and turned out to be very efficient for practical use, while keeping an almost linear worst case complexity. Furthermore only few data structures are needed, especially in contrast to the Martelli/Montanari algorithm. Its idea bases on theoretical framework developed by Huet (1976) and Paterson/Wegman (1978).
First of all we give two basic definitions:
A *homogeneous equivalence relation* is such that two nonvariable terms s and t are equivalent iff either one of them is a variable or they correspond to the same constant or function symbol (homogeneity condition) and their i-th subterms s_i and t_i are pairwise equivalent.
A *valid equivalence relation* is a homogeneous relation with a partial order on equivalence classes such that the class of t is before the class of t_i whenever t_i is a subterm of t.

Huet (1976) and Paterson/Wegman (1978) showed that two terms s and t are unifiable iff there exists a valid equivalence relation that makes s equivalent to t.

The algorithm is naturally decomposed in two steps:

Step1 (HERE) : build a homogeneous equivalence relation ≅ check for clashes.

Step2 (VERE) : build from step1 a valid equivalence relation ≅ check for cycles.

We will now look at both steps in more detail:

Step1:

Both terms s and t are parallely passed through and whenever a variable occurs it is bound to the corresponding subterm t_i or if it is already bound to a term t'_i, step1 is recursively applied to t_i and t'_i. The homogeneous equivalence relation is manifested in the variable bindings and can be represented by a directed graph G with the following properties:

- nodes in G are terms, one single node corresponds to each variable in s or t.
- each connected component of G corresponds to an equivalence class and contains at most one nonvariable term.
- for any variable node x in G: outdegree(x)≤1, and for a nonvariable node t: outdegree(t)=0, indegree(t)=1.

Thus for each variable x only one pointer r(x) is needed; initially r(x)=nil. Step1 stops with *clash* if the homogeneity condition fails at some point.

We now give a simplified algorithm for step1:

```
TERM-HERE(s,t):
   IF s and t are not identical variables or constant symbols
   THEN IF s is a variable
            THEN VAR-HERE(s,t)
            FI
   ELSIF t is a variable
           THEN VAR-HERE(t,s)
   ELSE let s=f(s1,...,sn), t=g(t1,...,tn)
           IF f≠g
           THEN exit (CLASH)
           ELSE FOR i:=1 TO K DO
                     TERM-HERE(si,ti)
   FI

VARE-HERE(x,t)
   IF r(x)=nil
   THEN r(u) ← t
   ELSIF t is a variable and r(t)=nil
          THEN r(t) ← x
   ELSE mark x,
```

```
        TERM-HERE(r(u),t);
        unmark u
    FI
```

Example: Given $s=f(h(x_1,x_2,x_3),h(x_6,x_7,x_8),x_3,x_6)$
and $t=f(h(g(x_4,x_5),x_1,x_2),h(x_7,x_8,x_6),g(x_5,a)x_5)$.
HERE(s,t) builds the following graph G: $x_3 \rightarrow x_2 \rightarrow x_1 \rightarrow g(x_4,x_5)$
and $x_7 \rightarrow x_8 \rightarrow x_6 \rightarrow x_5 \rightarrow a$ and $x_4 \rightarrow x_5$.

The function VARE-HERE mainly finds out to which equivalence class a particular variable belongs or defines a new class as the union of two equivalence classes. To reach quasilinear complexity the exact version of the Escalda/Ghallab algorithm uses a special UNION-FIND algorithm relying on the so called collapse and weight rules.

Step2:

The second step consists in checking the validity of the homogeneous equivalence relation built in step1 and in defining the unifier explicitly.

For each variable node of graph G (see step1) a new arc s(x) (substitutor) is created by the function VAR-VERE. It recursively substitutes the variables in the term bound to x (by G) and detects cycles in marking already visited variables. Here is a simplified algorithm not exploiting the efficiency of a UNIONFIND algorithm:

```
VAR-VERE(u)
    IF u is marked
    THEN exit(cycle)
    ELSE mark u;
        s(u) ← TERM-VERE(r(u));
        unmark u
    FI

TERM-VERE(t)
    IF t is a variable
    THEN VAR-HERE(t)
    ELSIF t is constant
    THEN return t
    ELSE let t=f(t1,...,tn)
        return f(TERM-VERE(t1),...,TERM-VERE(tn))
    FI
```

Given the graph G of the example of step1 we would get the following unifier:
$s(x_3)=s(x_2)=s(x_1)=g(a,a)$ and $s(x_7)=s(x_8)=s(x_6)=s(x_5)=s(x_4)=a$.

The complete unification algorithm consists in calling once TERM-HERE(s,t) and if it succeeds, calling VAR-VERE(u) for all variables u in s and t.
In our LISP-implementation we represented the equivalence relation on variables in two ways:

1) as *hash-tables*:, which ensures a direct access to every equivalence class and therefore results in a quasilinear complexity but entails a disproportionate administration amount for 'small' terms.
2) as *assoc-lists*:, which seems to be more efficient in practice, but at the expense of a quasilinear time complexity.

You can call both versions of the Escalada/Ghallab algorithm in TENUA by using the UNIFY function (see chapter 1.5).
Example: (UNIFY 'EG-HASH ⟨$term_1$⟩ ⟨$term_2$⟩) applies the first version to ⟨$term_1$⟩ and ⟨$term_2$⟩ while (UNIFY 'EG ⟨$term_1$⟩ ⟨$term_2$⟩) does the same for the second version.

3. Description of the Implemented Term Generators

To test unification algorithms TENUA provides two essentially different kinds of term generators:

(3.1) Generators for parameterized terms, say terms with a uniform structure
(3.2) Generators for random terms

Both kinds of term generating functions may get different parameters for input but return always a list of two terms s and t in list representation corresponding to the unification problem ⟨ s=t ⟩.

3.1 Parameterized Term Generators

With *parametrized terms* we mean terms with a uniform structure, also called *standard terms*. TENUA contains 8 generating functions for standard terms: GENTERM-STD1 ... GENTERM-STD8. They all get a nonnegative integer n for input and return terms with increasing size depending on n. In this way you can find out the time behaviour (exponential, linear,...) of a given unification algorithm. Notice, that for some of them the generated terms are growing non-linear in n.We give now a short description of each standard term generator:

GENTERM-STD1

Input: N: nonnegative integer

Value: A list of two terms of the following form:

1st term: $f(x_1,...x_n)$

2nd term: $f(y_1,...y_n)$

Application: Illustrates the behaviour of unification algorithms for increasing term breath.

GENTERM-STD2

Input: N: nonnegative integer

Value: A list containing two terms:

The first term has the form a left, the second of a right ZZ-tree of depth n, where ZZ-trees are defined as follows:

- a left ZZ-tree of depth 1 consists only of a root
- a right ZZ-tree of depth 1 consists only of a root
- a left ZZ-tree of depth k (>1) is a binary tree, where the left son is a right ZZ-tree of depth k-1 and the right son is a leave.
- a right ZZ-tree of depth k (>1) is a binary tree, where the right son is a left ZZ-tree of depth k-1 and the left son is a leave.
- the leaves are marked with a continuously indexed x respectively y.

Example: n = 5

1st term: $f(f(x_2,f(f(x_4,x_5),x_3)),x_1)$

2nd term: $f(y_1,f(f(y_3,f(y_5,y_4)),y_2)$

Application: Illustrates the behaviour of unification algorithms for variable bindings of increasing complexity, especially the occurcheck is tested.

GENTERM-STD3

Input: N: nonnegative integer

Value: A list containing two terms where each of them has the structure of a complete binary tree of depth n. The leaves are marked with continuously indexed variables x, respectively y.

Example: n = 5:

1st term: $f(f(f(x_1,x_2),f(x_3,x_4)),f(f(x_5,x_6),f(x_7,x_8)))$

2nd term: $f(f(f(y_1,y_2),f(y_3,y_4)),f(f(y_5,y_6),f(y_7,y_8)))$

Application: Illustrates the behaviour of unification functions for terms of increasing depth.

GENTERM-STD4

Input: N: nonnegative integer

Value: A list containing two terms, each of them possessing n arguments. The i-th argument corresponds to a term of GENTERM-STD2 with depth i. The toplevel functor is p and the leaves are marked with continuously indexed variables x, respectively y.

Application: Illustrates the behaviour of unification functions for terms of increasing depth and breadth and corresponding variable bindings.

GENTERM-STD5

Input: N: nonnegative integer

Value: A list of two terms of the following form:

1st term: $f(x_1,x_2,\ldots,x_n)$

2nd term: $f(x_2,x_3,\ldots,x_n,x_{n-1})$

Application: Tests the efficiency in finding variable bindings because all variables are bound to the same term. Here especially unification algorithms using UNION-FIND strategies will dominate.

GENTERM-STD6

Input: N: nonnegative integer

Value: A list of two terms of the following form:

1st term: $f(x_1,\ldots,x_n)$

2nd term: $f(g(x_0,x_0),\ldots,g(x_{n-1},x_{n-1}))$

Application: Classical example for the exponential complexity of the Robinson algorithm.

GENTERM-STD7

Input: N: nonnegative integer

Value: A list of two terms of the following form:

1st term: $f(x_1,\ldots,x_n,g(y_0,y_0),g(y_1,y_1),\ldots,g(y_{n-1},y_{n-1}))$

2nd term: $f(g(x_0,x_0),g(x_1,x_1),\ldots,g(x_{n-1},x_{n-1}),y_1,\ldots y_n)$

Application: This example is gathered from Bidoit/Corbin (1983) and shows that even the improved version of Robinson's algorithm keeps exponential time complexity.

GENTERM-STD8

Input: N: nonnegative integer

Value: A list of two terms of the following form:

1st term: $f(y_1, y_1,\ldots,y_n,y_n)$

2nd term: $f(x_1,g(x_0,x_0), x_2,g(x_1,x_1),\ldots,x_n,g(x_{n-1},x_{n-1}))$

Application: This is a modified version of GENTERM-STD6 and shows that unification algorithms (for example Robinson's) may have exponential time complexity, although the two terms have no variables in common.

3.2 Generators for Random Terms

First of all it must be said, that we did not want to consider the problem of generating a random termpair under the strict conditions of probability theory, but from a more practical viewpoint. The user should have the possibility to test the correctness and efficiency of unification algorithms by 'natural' terms, say terms that are not of extreme breadth or depth. The problem is not trivial, because generating both terms one after the other would lead to an extremely high quota of nonunifiable terms.

TENUA provides two generators for random term pairs: GENTERM-RND1 (see chapter 3.2.1) and GENTERM-RND2 (see chapter 3.2.2). They have only optional parameters (to adjust maximum termdepth, probability of clashes, etc.) and return a list of two terms in list representation. While GENTERM-RND1 allows the user to specify a signature of function symbols and then constructs the two terms in parallel, GENTERM-RND2 first generates a unifier and, out of that, the termpair, in order to support an aimed production of clashes, cycles and so on.

3.2.1 GENTERM-RND1

GENTERM-RND1 is a function which generates a term pair depending on several input arguments. These input parameters are optional. If they are not specified GENTERM-RND1 starts with some default values. We will now describe the input arguments and their influence on the construction of the term pair:

Input:	VAR-PROB (optional)	nonnegative integer ≤ 100 (probability for placing a variable instead of a function symbol)
	Default value:	45
	MAX-DEPTH (optional)	nonnegative integer (maximum depth of terms)
	Default value:	5
	CLASH-PROB (optional)	nonnegative integer ≤ 100 (probability of producing a clash)
	Default value:	30
	SIGNATURE (optional)	list containing elements of the form (F N M), where F is a function symbol, N is a nonnegative integer specifying the arity of F, and M is a nonnegative integer, specifying the relative frequency of F in relation to other function

symbols in SIGNATURE.
(signature of used function symbols; by specifying SIGNATURE the user can model his own practical conditions)

Default value: ((A 0 90) (F 1 20) (G 2 40) (H 3 20) (I 4 5) (R 5 3) (P 6 2) (S 10 1))
Here, for instance, the functors G and H have arity 2 respectively 3 but the the probability to be chosen is twice as high for G as for H.

Value: A list of two terms in list representation.

The construction of the term pair can be divided into three steps:

1) Parallel construction of a scheme of the two terms, but all function symbols (inclusded constants) are placed, but variables are not inserted yet, only their positions are marked.
2) Insertion of variables into the term schemes.
3) With probability VAR-PROB a clash-symbol is built in.

We will now describe the three steps seperately:

Step1: Construction of the scheme for both terms

Both term structures are constructed in parallel, such that no *direct clashes* (also called *structure clashes*)occur. The process can be outlined by the following algorithm:

```
TERMPAIR-STRUCTURE (MAX-DEPTH, SIGNATURE, VAR-PROB)
IF  MAX-DEPTH=1
THEN  return (VARMARK VARMARK)
ELSIF  RANDOM(100)≤VAR-PROB
THEN  return (VARMARK TERM-STRUCTURE (MAX-DEPTH -1, SIGNATURE))
ELSE  choose a function symbol F from SIGNATURE;
        return F(TERMPAIR-STRUCTURE (MAX-DEPTH -1, SIGNATURE,
                VAR-PROB),...,
                TERMPAIR-STRUCTURE (MAX-DEPTH -1,
                SIGNATURE, VAR-PROB))
```

where RANDOM(n) is a random number generator with range 1...n-1 VARMARK marks a position where a variable will be inserted (see Step2), and TERM-STRUCTURE is defined as follows:

```
TERM-STRUCTURE (MAX-DEPTH, SIGNATURE)
IF  MAX-DEPTH=1
THEN return VARMARK
```

ELSE choose a function symbol F from SIGNATURE;
return F (TERM-STRUCTURE (MAX-DEPTH -1, SIGNATURE),...
...,TERM-STRUCTURE (MAX-DEPTH -1, SIGNATURE))

Step2: Insertion of variables into the term schemes

Let N be the number of variables in both term schemes. We define the set of variables VARS as $\{x_1, x_2,\ldots,x_N\}$. Now for every marked position a variable is randomly chosen from VARS and inserted into that position. It may happen that the same variable occurs more than once, so that indirect clashes are possible.

Step3: Insertion of clashes

In step3 a clash-symbol is built into the termpair (resulting from step1 and step2) with probability CLASH-PROB. For that purpose a node is choosen from both terms. If a leave is choosen, it is marked with the symbol B0815 else if an inner node is chosen it is marked with symbol T4711. This procedure doesn't guarantee a clash since the corresponding node may be a variable.

To simplify the specification of a signature TENUA provides some default signatures, which are bound to the following LISP constants:

MONOID ≈ ((M 2 20) (E 0 10))
GRUPPE ≈ ((M 2 20) (E 0 10) (I 1 20))
RING ≈ ((M 2 20) (P 2 20) (E1 0 10) (E2 0 10) (I 1 20))
KOERPER ≈ ((M 2 20) (P 2 20) (E1 0 10) (E2 0 10) (I1 1 20) (I2 1 20))

Possible calls of GENTERM-RND1 would be:
(GENTERM-RND1) (maintenance of all default values) or
(GENTERM-RND1 45 6 20 GRUPPE) (term pairs of group will be generated with maximal depth 6 and clash probability 20).

Conclusion:

It becomes clear that the term pair is constructed in view of a practical unification application and less under the aspect of probability theory: the quota of unifiable terms is disproportionately high, there are even term pairs which cannot be generated (for example multiple clashes), GENTERM-RND1 therefore is not surjective. However in practice this may not be of high interest.

On the other hand in specifying SIGNATURE the user has the possibility of modelling his own practical conditions and can adjust term size, clash probability and variable occurrences according to his requirements.

3.2.2 GENTERM-RND2

GENTERM-RND2, like GENTERM-RND1, is a function that generates a random term pair depending on several optional input arguments. Unlike GENTERM-RND1 the terms are not immediately constructed. Instead their construction is guided by a substitution that has been generated before. By appropriate manipulations of the substitution a specific generation of clashes, cycles etc. is possible. We will describe the different optional parameters and their influence on the construction of the term pair:

Input: (all input parameters are optional)

VARLIST — List of variables from which the variables in the term pair are taken.
Default value: (X Y Z)

UNI-PROB — nonnegative integer ≤ 100;
probability for producing a unifiable termpair
Default value: 50

DCL-PROB — nonnegative integer ≤ 100;
probability for producing a direct clash ('structure clash')
Default value: 15

ICL-PROB — nonnegative integer ≤ 100;
probability for producing an indirect clash
Default value: 5

CYC-PROB — nonnegative integer ≤ 100;
probability for producing a cycle
Default value: 14

CYC-DCL-PROB — nonnegative integer ≤ 100;
probability for producing a pair containing a direct clash and a cycle
Default value: 4

D/ICL-PROB — nonnegative integer ≤ 100;
probability for producing a direct and an indirect clash
Default value: 4

CYC-ICL-PROB nonnegative integer ≤ 100;
probability for producing a pair containing an indirect clash and a cycle

Default value: 4

CYC-D/ICL-PROB nonnegative integer ≤ 100;
probability for producing a termpair containing an indirect clash, a direct clash and a cycle

Default value: 4

Restriction: UNI-PROB + DCL-PROB + ICL-PROB + CYC-PROB + CYC-DCL-PROB + D/ICL-PROB + CYC-ICL-PROB + CYC-D/ICL-PROB = 100

Exception: If starting from a certain parameter all following parameters are equal to 0, they need not be specified.

MAX-DEPTH-TERMSTRUCTURE nonnegative integer;
maximum depth of the term structure into which the unifier is built

Default value: 8

MAX-DEPTH-UNI nonnegative integer;
maximum depth of the terms in the unifier

Default value: 4

SMALL-TERM-PROB nonnegative integer ≤ 100;
probability for stopping the construction of a term (see step1)

Default value: 70

CONST-PROB nonnegative integer ≤ 100;
probability for placing a constant instead of a variable

Default value: 40

CLASH-SYM-PROB nonnegative integer ≤ 100;
probability for substituting function symbols by a

clash symbol, supposed a direct clash is to be produced.

GENTERM-RND2 works with a fixed signature of function symbols: A (constant or 0-ary function symbol), F1 (1-ary function symbol), F2 (2-ary function symbol), and so on. The clash symbols are analogously named by B, G1, G2....
The construction of the pair can be divided into 4 steps:
1) Constrution of the unifier
2) Construction of a pair of identical term schemes
3) A direct clash is built into both term structures with probability DCL-PROB
4) Insertion of the unifier into the leaves of both term structures
We will now discuss the different steps in more detail:
Step 1: Construction of the unifier

Constructing a unifier means that every variable of VARLIST must be bound to a term. It may happen that several variables are bound to the same term. Therefore VARLIST first is first partitioned, such that all variables of the same subset are bound to the same term. The terms corresponding to the different subsets are generated by the following method:
a) First the basic term structure is generated:
Starting with a root as actual node, it is decided at every step if a new leave is added to the actual node (probability 100 - SMALL-TERM-PROB), or if the father becomes the new actual node (probability SMALL-TERM-PROB). The process stops if the father of the root is demanded. Moreover it must be guaranteed that the term structure is not deeper than MAX-DEPTH-UNI.
b) Now the term structure generated in a) is labeled in the following way:
Inner nodes are labeled with the corresponding function symbol and leaves are marked either with a constant symbol (probability CONST-PROB), or with a variable symbol (probability 100 - CONST-PROB).

Before the unifier is constructed it must be decided what kind of term pair (unifiable, cycle...) shall be generated, because this will influence the construction of the unifier.
If two unifiable terms are to be generated (probability UNI-PROB), cycles must be excluded. Therefore a term corresponding to a certain variable subset must contain only variables belonging to the subsequent subsets (Occur-Check).
In case a cycle is to be (probability CYC-PROB) the restriction on variables is canceled, moreover the number of subsets is reduced and the probability that variables are created instead of constants is increased. This guarantees a cycle quota of about 98%. Nevertheless even if CYC-PROB is 100 a termpair may arise which does not contain cycles.
If an indirect clash is to be produced (probability ICL-PROB), first the unifier is

constructed analogously to the unifiable case. Then it is enlarged by multiple bindings of some variables. The terms belonging to these additional bindings may contain clash symbols.

In case of a mix of a cycle and an indirect clash the unifier is constructed as in the cycle case, but subsequently enlarged by additional variable bindings.

Direct clashes are inserted later (see step3).

One may ask why some features are produced in a rather complicated way (for example cycles). This was necessary because we wanted to ensure that any possible term pair can be produced by GENTERM-RND2 (surjectivity).

Step2: Construction of a pair of identical term schemes

The task is to generate a term scheme, i.e. a term with unlabeled leaves, and then to copy it. The term scheme is constructed analogously to step1, including the following points:

- The term structure must not be deeper than MAX-DEPTH-TERMSTRUCTURE.
- The number of (unmarked) leaves corresponds to the number of variable bindings of the unifier (including multiple bindings), such that every leave corresponds to a pair consisting of a variable and a term.

This is guaranteed by a new stop criterion:

Stop the construction if the required number of leaves is reached.

Step3: Eventually direct clash symbols are built in:

If a direct clash, the union of a direct clash and a cycle, an indirect clash is to be produced, in step1the unifier is constructed without considering the direct clash.

If for instance a direct clash and a cycle are required, the unifier is constructed as in the cycle case. Then step2 is performed and only (in step3) clash symbols are built into the second term structure, substituting every label by the corresponding clash symbol with probability CLASH-SYM-PROB.

Step4: Insertion of the unifier into the leaves of both term structures

As mentioned above, every pair of corresponding leaves of the two identical termstructures of step2 belongs to a pair of the form (variable term). The leaves are labeled in the following way: Given such pair of leaves and its corresponding (VAR TERM) either the leave in the first term structure or that in the second one is labeled with VAR. The remaining unmarked leave is either labeled with TERM, or with a variable belonging to the same subset as VAR. However every term, bound to a subset of variables, must occur at least once. In this way indirect variable bindings are possible.

Possible calls of GENTERM-RND2 are:

(GENTERM-RND2)	(default values for all arguments)
(GENTERM-RND2 '(X Y Z) 100))	(a unifiable pair of terms is produced)
(GENTERM-RND2 '(X Y Z) 0 0 0 0 100)	(a pair containing a clash and a cycle is produced).

Conclusion:
Like GENTERM-RND1, GENTERM-RND2 is designed to meet practical requirements. Its precise production of clashes, cycles etc. makes a thorough analysis of unification algorithms possible. I.e. one can determine the time complexity of detecting cycles or find out what is detected first by different algorithms, an indirect clash or a cycle. It is helpful that the performance in the different cases can be tested seperately. In addition to this see also chapter 4 on statistical functions, which allow a comfortable handling with a great numbersof examples. Another advantage of GENTERM-RND2 is its surjectivity: every possible termpair can be generated. The user can even manipulate the probability of certain classes of terms.

4. Facilities for Testing Correctness and Efficiency of Unification Algorithms

4.1 Facilities for Testing Correctness of Unification Algorithms

Before the efficiency of a newly implemented unification algorithm is measured, its correctness must be checked. For this purpose TENUA provides a function CORRECTNESS, that performs a test on equivalence of mgu's:

Input:	TERM-PAIR	list of two terms in list representation
	UNIFY-ALGO	function, which gets a term pair in list representation and returns CLASH, CYCLE or an explicit unifier of the input terms represented as assoc-list
Value:	T	if the given term pair is unifiable and UNIFY-ALGO returns an mgu of it, or if the term pair is not unifiable and UNIFY-ALGO returns CLASH or CYCLE
	<error message>	otherwise

Example: The call (CORRECTNESS (GENTERM-RND2 '(X Y Z) 100) 'NEW-UNIFY) causes NEW-UNIFY to be applied to a unifiable term pair generated by GENTERM-RND2, and it is checked if the result is a most general unifier.

You can perform a great number of tests by using the function DOTIMES .

Example: (DOTIMES (K 100) (CORRECTNESS (GENTERM-RND2 '(X Y Z) 100) 'NEW-UNIFY)

repeats the above test 100 times. Notice, that DOTIMES return NIL. Your algorithm is wrong, if error messages occur!

4.2 Facilities for Testing Efficiency of Unification Algorithms

Most people working with unification algorithms are interested in their time comlexity. Either they want to measure the runtime of a special algorithm for some kinds of terms, or they want to compare different algorithms with regard to some application. Often it is also useful to find out, how some unification algorithms work: whether it detects first clashes or cycles?....To make the answers to these questions easier, TENUA provides statistical functions that use the term generators discussed in chapter 3. We think they constitute the most important part of TENUA. There are two kinds of statistical functions:

- statistical functioms using the standard term generators (see 4.1.1)
- statistical functions using the random term generators (see 4.1.2).

4.2.1 Efficiency Test Using Standard Terms

You can test the performance of unification algorithms on standard terms in TENUA using the function STATISTIC-GENTERM-STD. It allows a comfortable work with the standard term generators discussed in chapter 3, and so an analysis of time complexity at systematically varying terms. STATISTIC-GENTERM-STD has the following input/output specification:

Input:

TERMGEN	one of the standard term generators GENTERM-STD1-8
FROM	nonnegative integer
TO	nonnegative integer
STEP	nonnegative integer
FUN (key)	list, containing the names of unification functions (for example ROB, EG,MM)

FOREIGN-FUN (key) list, containing the names of unification functions, which are to be measured without considering the time used for term and unifier conversion. For this reason the pure unification function must be enclosed by the function STATISTIC-TIME.

Example:

```
(DEFUN USER-UNIFY-HELP (Term1 Term2)
     (LET*
       ((T1 (CONVERT-TERM Term1))
        (T2 (CONVERT-TERM Term2))
        (HELP (STATISTIC-TIME(USER-UNIFYT1T2)))
       (LIST (FIRST HELP)
             (CONVERT-SUBST (SECOND HELP)))))
```

OUTFILE (key) string, specifying the name of a file

<u>Value</u>: The unification functions specified in FUN and FOREIGN-FUN are applied to the term pairs that arise from TERMGEN. TERMGEN starts with input argument FROM and stops with TO making steps of width STEP. At every step the runtime of the different unification functions is measured; the functions specified in FOREIGN-FUN without considering input/output conversion. The results are displayed on screen in table form or, if OUTFILE was specified, are additionally written there.

<u>Example</u>:

1) The call

```
(STATISTIC-GENTERM-STD 'GENTERM-STD6 1 10 1
                       :FUN '(ROB EG-HASH)
                       :FOREIGN-FUN '(MM-OHNE-KONVERT)
                       :OUTFILE "TEST")
```

performs the unification functions ROB, EG-HASH and MM-OHNE-KONVERT (MM without input/output conversion) on GENTERM-STD6 and writes the following results on screen and on file TEST:

Number of tests: 10
Start Value: 1
Stop Value: 10
Step Width: 1

N	ROB	EG-HASH	MM-OHNE-KONVERT
1	0.0035	0.0137	0.0068
2	0.0069	0.0228	0.0152
3	0.0192	0.0256	0.0201
4	0.0343	0.0341	0.0257
5	0.0734	0.0371	0.0341
6	0.1477	0.0460	0.0461
7	0.2929	0.0522	0.0460
8	0.5937	0.0629	0.0549
9	1.1882	0.0638	0.0610
10	2.5781	0.0690	0.0713

This example illustrates the exponential time complexity of the Robinson algorithm in contrast to

the linear respective quasilinear time behaviour of the Escalada/Ghallab and the Martelli/Montanari algorithm.

2) Suppose you want to analyze the time complexity of USER-UNIFY (see above) for growing term breadth without measuring time for input/output conversion, the appropriate call is:

```
(STATISTIC-GENTERM-STD 'GENTERM-STD1 5 100 5
                        :FOREIGN-FUN 'USER-UNIFY-HELP).
```

4.2.2 Efficiency Test Using Random Terms

The function STATISTIC-GENTERM-RND allows a comfortable use of the random term generators described in chapter 3. In this way the user has the possibility to analyze the performance of unification algorithms under such conditions as: special classes of termpairs (unifiable, clash, cycle...), different signatures and so on. STATISTIC-GENTERM-RND has the following input/output specification:

Input:	LOOP-NUM	nonnegative integer
	TERMGEN-CALL	function call of GENTERM-RND1 or GENTERM-RND2
	FUN (key)	list of unification functions
	FOREIGN-FUN (key)	list of unification functions which shall be measured without input/output conversion (see chapter 4.2.1)
	PROTOKOLL (key)	T(rue) or F(alse)
	OUTFILE (key)	string, specifying the name of a file

Value: TERMGEN-CALL is performed LOOP-NUM times and the different unification functions stated in in FUN and FOREIGN-FUN are applied to every produced term pair. If you want to record every generated term pair along with its unifier, you must set PROTOKOLL to T(rue). Now the run times of every unification function are divided into the classes UNIFIABLE, CLASH, CYCLE and some statistical quantities are calculated. The results are printed on screen or, if OUTFILE was specified, they are additionally written there.

Example: Suppose you want to analyze the behaviour of the Robinson and the Escalada/Ghallab algorithm and you are especially interested in the question what is detected first by which algorithm: an indirect clash or a cycle? Then an appropriate call of STATISTIC-GENTERM-RND is:

```
(STATISTIC-GENTERM-RND 100
             '(GENTERM-RND2 '(X Y Z) 0 0 0 0 0 0 100)
             :FUN '(ROB EG)).
```

TENUA prints the following results on screen:

TENUA prints the following results on screen:

Number of generated Termpairs:	100
Average Length of Terms:	11.43
Standard Deviation:	3.10
Maximum Length:	35
Minimum Length:	5

Function	Number		Time		Time/Number
ROB	absolut	relative	absolut	relative	
CLASH	3	3.0%	0.024	4.1%	0.0081
CYCLE	97	97.0%	0.573	95.9%	0.0059
UNIFIABLE	0	0.0%	0.0	0.0%	0.0
Σ	100	100.0%	0.597	100.0%	0.0060

Function	Number		Time		Time/Number
EG	absolut	relative	absolut	relative	
CLASH	88	88.0%	0.451	80.6%	0.0051
CYCLE	12	12.0%	0.108	19.4%	0.0090
UNIFIABLE	0	0.0%	0.0	0.0%	0.0
Σ	100	100.0%	0.559	100.0%	0.0056

Now it is easy to reach the following conclusion: Given a mix of cycle and indirect clash the Robinson algorithm normally detects the cycle first, while the Escalada/Ghallab algorithm usually finds the clash.

5. Conclusion

A test environment for unification algorithms has been presented. Providing input/output interface, some well known implemented unification algorithms, generators for random and parameterized term pairs, and finally statistical functions, it turns out to be a useful tool for the implementation as well as for the analysis of unification algorithms. Especially the possibility to compare the time complexity of different algorithms with regard to certain classes of termpairs is of practical interest. With the implemented unification functions we made the following experiences (see appendix): Although its worst case complexity is exponential the Robinson

algorithm (ROB) has excellent performances; especially for small terms. The Martelli/Montanari algorithm (MM) turned out to be the slowest of the implemented algorithms.This is caused by complex data structures the initialization and management of which worsens significantly the average performance for small terms. Especially the creation and initialization of the hash tables, that represent the equivalence classes of variables, seems to be very expensive. This conjecture was confirmed when omitting the time for input/output conversion (MM-OHNE-KONVERT). Indeed the pure time, needed for unification by the Martelli/Montanari algorithm is highly competitive. Thus it will be especially attractive for systems which already use an analogous representation of terms. In contrast to this the Escalada/Ghallab algorithm (EG-HASH) involves very simple data structures. This explains its outstanding performances for small terms while keeping a almost linear worst case complexity. Substituting the hash table representation of substitutions by association lists (EG) it will be as twice as quick, comparable to the Robinson algorithm even for small terms, but, at the cost of a $O(n^2)$ complexity. We think that it is worth to be tested in practice more frequently. As a final remark, we point out, that an extension of TENUA to unification with equations might be possible by additionally implementing appropriate unification algorithms and standard term generators. The statistical functions also might be improved by using a graphical representation.

6. References

1. J. Corbin and M. Bidoit:
   A Rehabilitaion of Robinson's Unification Algorithm, Information Processing, 1983.
2. G. Escalada and M. Ghallab:
   A Practically Efficient and Almost Linear Unification Algorithm, Laboratoire d'Automatique et d'Analyse des Systèmes, Toulouse, 1987.
3. J.Herbrand:
   Recherches sur la Theorie de la Demonstration, Thesis, Paris 1930
4. A. Martelli and U. Montanari:
   An Efficient Unification Algorithm, Journal of the ACM, 1982.
5. M.S. Paterson and M.N. Wegman:
   Linear Unification, Journal of Computer and System Sciences Vol.16, 1978.
6. J.A. Robinson:
   A Machine-Oriented Logic Based on the Resolution Principle, Journal of the ACM Vol.12, 1965.
7. J.A. Robinson:
   Computational Logic - the Unification Computation, Machine Intelligence Vol.6, 1971

Vergleich von Unifikations-Funktionen mittels
Standardterm-Generator : GENTERM-STD1

Anzahl der Testlaeufe : 10
Startwert : 10
Ende : 100
Schrittweite : 10

N	ROB	MM	EG	EG-HASH	MM-OHNE-KONVERT
10	0.0417	0.1685	0.0557	0.0565	0.0423
20	0.0934	0.2042	0.0709	0.0788	0.1014
30	0.1950	0.3656	0.1280	0.1450	0.1841
40	0.3390	0.5046	0.1971	0.1979	0.2691
50	0.5192	0.7315	0.2818	0.2805	0.3837
60	0.7350	0.9018	0.4018	0.3206	0.5112
70	0.9858	1.1001	0.4921	0.3648	0.6602
80	1.2678	1.3238	0.6217	0.3921	0.8240
90	1.7043	1.7286	0.7609	0.5601	0.9945
100	2.0627	1.9228	0.9604	0.5860	1.1921

Vergleich von Unifikations-Funktionen mittels
Standardterm-Generator : GENTERM-STD2

Anzahl der Testlaeufe : 10
Startwert : 2
Ende : 20
Schrittweite : 2

N	ROB	MM	EG	EG-HASH	MM-OHNE-KONVERT
2	0.0026	0.0253	0.0082	0.0146	0.0067
4	0.0126	0.0471	0.0063	0.0198	0.0183
6	0.0130	0.0716	0.0082	0.0217	0.0270
8	0.0196	0.0938	0.0101	0.0241	0.0372
10	0.0261	0.1146	0.0180	0.0262	0.0504
12	0.0298	0.1398	0.0167	0.0284	0.0648
14	0.0366	0.1667	0.0157	0.0306	0.0802
16	0.0404	0.1917	0.0204	0.0328	0.0894
18	0.0466	0.2170	0.0223	0.0349	0.1045
20	0.0532	0.2680	0.0242	0.0373	0.1192

Vergleich von Unifikations-Funktionen mittels
Standardterm-Generator : GENTERM-STD3

Anzahl der Testlaeufe : 10
Startwert : 1
Ende : 10
Schrittweite : 1

N	ROB	MM	EG	EG-HASH	MM-OHNE-KONVERT
1	0.0006	0.0012	0.0026	0.0149	0.0000
2	0.0026	0.0256	0.0047	0.0175	0.0096
3	0.0113	0.0447	0.0094	0.0237	0.0168
4	0.0265	0.0852	0.0232	0.0369	0.0338
5	0.0703	0.1728	0.0520	0.0644	0.0815
6	0.2293	0.4131	0.1449	0.1531	0.2011
7	0.4516	1.0420	0.4288	0.3400	0.5866
8	1.1175	2.8167	1.4386	0.7201	1.8689

Vergleich von Unifikations-Funktionen mittels
Standardterm-Generator : GENTERM-STD4

Anzahl der Testlaeufe : 10
Startwert : 2
Ende : 20
Schrittweite : 2

N	ROB	MM	EG	EG-HASH	MM-OHNE-KONVERT
2	0.0054	0.0362	0.0070	0.0207	0.0103
4	0.0298	0.1160	0.0240	0.0400	0.0508
6	0.0926	0.3046	0.0482	0.0602	0.1290
8	0.1694	0.5246	0.0715	0.0874	0.2426
10	0.2707	0.8742	0.1152	0.1261	0.4475
12	0.4397	1.3427	0.1604	0.1602	0.7645
14	0.6559	2.0504	0.2279	0.2102	1.1899
16	0.9921	2.9609	0.3319	0.2636	1.8193
18	1.4103	4.0835	0.4037	0.3288	2.7135
20	1.8072	5.6497	0.5240	0.4144	3.8390

Vergleich von Unifikations-Funktionen mittels
Standardterm-Generator : GENTERM-STD5

Anzahl der Testlaeufe : 10
Startwert : 2
Ende : 20
Schrittweite : 2

N	ROB	MM	EG	EG-HASH	MM-OHNE-KONVERT
2	0.0024	0.0229	0.0053	0.0182	0.0089
4	0.0088	0.0361	0.0133	0.0227	0.0166
6	0.0135	0.0533	0.0188	0.0298	0.0225
8	0.0195	0.0718	0.0249	0.0456	0.0355
10	0.0267	0.0900	0.0318	0.0511	0.0506
12	0.0381	0.1132	0.0384	0.0575	0.0643
14	0.0487	0.1356	0.0485	0.0686	0.0781
16	0.0590	0.1600	0.0563	0.0754	0.0968
18	0.0751	0.1855	0.0649	0.0855	0.1171
20	0.1117	0.2685	0.0825	0.0943	0.1405

Vergleich von Unifikations-Funktionen mittels
Standardterm-Generator : GENTERM-STD6

Anzahl der Testlaeufe : 10
Startwert : 1
Ende : 10
Schrittweite : 1

N	ROB	MM	EG	EG-HASH	MM-OHNE-KONVERT
1	0.0030	0.0225	0.0030	0.0120	0.0084
2	0.0054	0.0342	0.0067	0.0204	0.0134
3	0.0146	0.0442	0.0134	0.0236	0.0187
4	0.0328	0.0578	0.0146	0.0374	0.0244
5	0.0690	0.0710	0.0188	0.0370	0.0327
6	0.1740	0.1020	0.0302	0.0520	0.0511
7	0.2922	0.0920	0.0306	0.0483	0.0441
8	0.6333	0.1398	0.0461	0.0537	0.0528
9	1.2664	0.1271	0.0375	0.0649	0.0589
10	2.6121	0.1326	0.0434	0.0640	0.0686

Vergleich von Unifikations-Funktionen mittels
Standardterm-Generator : GENTERM-STD7

Anzahl der Testlaeufe : 10
Startwert : 1
Ende : 10
Schrittweite : 1

N	ROB	MM	EG	EG-HASH	MM-OHNE-KONVERT
1	0.0099	0.0494	0.0179	0.0325	0.0129
2	0.0355	0.0649	0.0240	0.0432	0.0263
3	0.0800	0.0835	0.0391	0.0592	0.0367
4	0.1931	0.1144	0.0549	0.0763	0.0510
5	0.4418	0.1392	0.0668	0.0904	0.0654
6	1.0045	0.1588	0.0822	0.1057	0.0809
7	2.3387	0.1823	0.1035	0.1213	0.0943
8	5.2158	0.2091	0.1195	0.1356	0.1406
9	11.0254	0.4345	0.1844	0.1739	0.1316
10	24.1122	0.5290	0.2038	0.1853	0.1497

Vergleich von Unifikations-Funktionen mittels
Standardterm-Generator : GENTERM-STD8

Anzahl der Testlaeufe : 10
Startwert : 1
Ende : 10
Schrittweite : 1

N	ROB	MM	EG	EG-HASH	MM-OHNE-KONVERT
1	0.0034	0.0307	0.0066	0.0173	0.0118
2	0.0100	0.0507	0.0142	0.0314	0.0208
3	0.0297	0.0813	0.0253	0.0544	0.0434
4	0.0657	0.0880	0.0346	0.0527	0.0404
5	0.1391	0.1085	0.0437	0.0635	0.0535
6	0.2891	0.1319	0.0535	0.0752	0.0674
7	0.5974	0.1618	0.0643	0.0902	0.0800
8	1.3152	0.1757	0.0780	0.0969	0.0948
9	2.6702	0.2378	0.0928	0.1156	0.1192
10	4.9942	0.2425	0.1104	0.1355	0.1411

Vergleich von Unifikations-Funktionen mittels
Zufallsterm-Generator : (GENTERM-RND1)

Anzahl der erzeugten Terme :	1000
Durchschnittliche Termgroesse :	7.66
Standard-Abweichung :	5.23
Laenge des groessten Termes :	68
Laenge des kleinsten Termes :	2

Funktion: ROB	Anzahl (absolut)	Anzahl (relativ)	Zeit (absolut)	Zeit (relativ)	Zeit/Anzahl
Clashes	256	25.6 %	1.930	19.2 %	0.0075
Zyklen	162	16.2 %	2.106	20.9 %	0.0130
Unif. Terme	582	58.2 %	6.020	59.9 %	0.0103
Summe	1000	100.0 %	10.056	100.0 %	0.0101

Funktion: MM	Anzahl (absolut)	Anzahl (relativ)	Zeit (absolut)	Zeit (relativ)	Zeit/Anzahl
Clashes	277	27.7 %	5.717	16.4 %	0.0206
Zyklen	141	14.1 %	6.492	18.7 %	0.0460
Unif. Terme	582	58.2 %	22.595	64.9 %	0.0388
Summe	1000	100.0 %	34.804	100.0 %	0.0348

Funktion: EG	Anzahl (absolut)	Anzahl (relativ)	Zeit (absolut)	Zeit (relativ)	Zeit/Anzahl
Clashes	279	27.9 %	0.810	11.8 %	0.0029
Zyklen	139	13.9 %	1.024	15.0 %	0.0074
Unif. Terme	582	58.2 %	4.999	73.2 %	0.0086
Summe	1000	100.0 %	6.833	100.0 %	0.0068

Funktion: EG-HASH	Anzahl (absolut)	Anzahl (relativ)	Zeit (absolut)	Zeit (relativ)	Zeit/Anzahl
Clashes	279	27.9 %	3.007	17.5 %	0.0108
Zyklen	139	13.9 %	2.424	14.1 %	0.0174
Unif. Terme	582	58.2 %	11.781	68.4 %	0.0202
Summe	1000	100.0 %	17.212	100.0 %	0.0172

Funktion: MM-OHNE-KONVERT	Anzahl (absolut)	Anzahl (relativ)	Zeit (absolut)	Zeit (relativ)	Zeit/Anzahl
Clashes	277	27.7 %	1.367	14.4 %	0.0049
Zyklen	141	14.1 %	1.895	20.0 %	0.0134
Unif. Terme	582	58.2 %	6.204	65.5 %	0.0107
Summe	1000	100.0 %	9.466	100.0 %	0.0095

Vergleich von Unifikations-Funktionen mittels
Zufallsterm-Generator : (GENTERM-RND2)

Anzahl der erzeugten Terme :	1000
Durchschnittliche Termgroesse :	7.16
Standard-Abweichung :	2.87
Laenge des groessten Termes :	77
Laenge des kleinsten Termes :	4

Funktion: ROB	Anzahl (absolut)	Anzahl (relativ)	Zeit (absolut)	Zeit (relativ)	Zeit/Anzahl
Clashes	278	27.8 %	0.747	7.7 %	0.0027
Zyklen	192	19.2 %	5.170	53.5 %	0.0269
Unif. Terme	530	53.0 %	3.751	38.8 %	0.0071
Summe	1000	100.0 %	9.668	100.0 %	0.0097

Funktion: MM	Anzahl (absolut)	Anzahl (relativ)	Zeit (absolut)	Zeit (relativ)	Zeit/Anzahl
Clashes	336	33.6 %	4.094	17.3 %	0.0122
Zyklen	134	13.4 %	4.394	18.5 %	0.0328
Unif. Terme	530	53.0 %	15.239	64.2 %	0.0288
Summe	1000	100.0 %	23.727	100.0 %	0.0237

Funktion: EG	Anzahl (absolut)	Anzahl (relativ)	Zeit (absolut)	Zeit (relativ)	Zeit/Anzahl
Clashes	336	33.6 %	0.629	11.1 %	0.0019
Zyklen	134	13.4 %	0.586	10.3 %	0.0044
Unif. Terme	530	53.0 %	4.463	78.6 %	0.0084
Summe	1000	100.0 %	5.677	100.0 %	0.0057

Funktion: EG-HASH	Anzahl (absolut)	Anzahl (relativ)	Zeit (absolut)	Zeit (relativ)	Zeit/Anzahl
Clashes	336	33.6 %	2.912	18.9 %	0.0087
Zyklen	134	13.4 %	1.806	11.7 %	0.0135
Unif. Terme	530	53.0 %	10.664	69.3 %	0.0201
Summe	1000	100.0 %	15.381	100.0 %	0.0154

Funktion: MM-OHNE-KONVERT	Anzahl (absolut)	Anzahl (relativ)	Zeit (absolut)	Zeit (relativ)	Zeit/Anzahl
Clashes	336	33.6 %	0.923	13.3 %	0.0027
Zyklen	134	13.4 %	1.115	16.1 %	0.0083
Unif. Terme	530	53.0 %	4.905	70.6 %	0.0093
Summe	1000	100.0 %	6.943	100.0 %	0.0069