MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 667

April 1982

Reasoning Utility Package
User's Manual
Version One

David Allen McAllester

Abstract: RUP (Reasoning Utility Package) is a collection of procedures for performing various computations relevant to automated reasoning. RUP contains a truth maintenance system (TMS) which can be used to perform simple propositional deduction (unit clause resolution), to record justifications, to track down underlying assumptions, and to perform incremental modifications when premises are changed. This TMS can be used with an automatic premise controller which automatically retracts "assumptions" before "solid facts" when contradictions arise and searches for the most solid proof of an assertion. RUP also contains a procedure for efficiently computing all the relevant consequences of any set of equalities between ground terms. A related utility computes "substitution simplifications" of terms under an arbitrary set of unquantified equalities and a user defined simplicity order. RUP also contains demon writing macros which allow one to write PLANNER like demons that trigger on various types of events in the data base. Finally there is a utility for reasoning about partial orders and arbitrary transitive relations. In writing all of these utilities an attempt has been made to provide a maximally flexible environment for automated reasoning.

Keywords: Reasoning Utility Package, Theorem Proving, Automated Deduction, Truth Maintenance, Backtracking, Dependencies, Assumptions, Congruence Closures, Demonic Invocation, Transitive Relations, Term Simplification.

This report describes work done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643 and in part by National Science Foundation Grant MCS77-04828.

Acknowledgments: There are many people who have contributed to the current RUP environment by making helpful suggestions and criticisms. I would especially like to thank Dan Brotsky, Ed Barton, Charles Rich, Ken Forbus, William Long, Lowell Hawkinson, and David Chapman for their criticisms and experimentation in using RUP.

CONTENTS

1. INTRODUCTION

RUP (Reasoning Utility Package) is a collection of utilities relevant to automated reasoning. RUP contains a truth maintenance system (TMS) which can be used to perform simple propositional deduction (unit clause resolution), to record justifications, to track down underlying assumptions, and to perform incremental modifications when premises are changed. RUP also provides a fast system for performing deductions concerning equalities. The equality system contains routines which "intern" expressions. This system also performs all deductions which can be made purely via substitution of equals for equals and can simplify terms under a large class of simplicity orderings. RUP also contains mechanisms for writing PLANNER-like demons. The demons created via this package can be compiled as ordinary lisp functions in which the pattern matching mechanism is open coded into the definition of each such demonic function.

In designing the RUP environment an attempt has been made to maximize the flexibility of the utilities and allow them to interact effectively with user defined systems. Thus there are many "hooks" which allow the user to modify RUP in different ways. For example hooks are provided for installing user defined backtracking functions and user defined pattern directed invocation mechanism. There is also a general control methodology adopted in RUP which associates queues with invariants. The demonic triggering mechanisms provided by RUP allow the user to define his own queues and invariants and to maintain those invariants by having forms queued demonically when an invariant is violated. RUP provides a simple data base in the form of interned expressions but users typically define their own data structures and define invariants which associate their data structures with those provided by RUP.

This document describes the major functions in RUP and examples of their use. The description of each function is prefaced by the name of the function in bold letters followed by the list of arguments taken by that function. RUP is implemented in both LISP Machine LISP and in MACLISP. There are two versions of the TMS one which implements a semi-automatic certainty based premise controller and one which leaves premise control entirely to the user. One can load RUP into LISP by loading whichever of the following files is appropriate (the files reside on MIT-AI):

AI:RUP;RUP >	LISP machine RUP without premise controller
AI:RUP;RUPP >	LISP machine RUP with semi-automatic premise control
AI:RUP;MRUP >	MACLISP RUP without premise controller
AI:RUP;MRUPP >	MACLISP RUP with semi-automatic premise control

The LISP machine versions loads into the package RUP and all symbols in this manual which do not have an explicit package prefix reside in the RUP package. In the MACLISP version package prefixes are simply interpreted as part of the character name.

2. SOME SIMPLE SCENARIOS

This section is intended for first time users who want to use RUP in the most straightforward manner possible. A series of scenarios is presented each of which is intended to demonstrate some feature of the top level RUP environment. The reader should be cautioned against just reading these scenarios and not reading the remainder of the manual. There are many utilities which are not demonstrated in these scenarios. Furthermore the scenarios emphasize the use of RUP as a programming language and leave out the important view of RUP as a utility package.

The first scenario demonstrated the simple propositional reasoning facilities and the explanation generation mechanisms.

The second scenario demonstrates how the simple propositional deduction mechanisms can be extended with a refutation mechanism invoked by the top level function **try-to-show** while the third scenario shows the

---

Scenario 1.

```
(assert '(:-> p q))
 ...
(assert '(:-> q r))
 ...
(assert 'p)
 ...
(why 'r)
"R IS :TRUE FROM:"
"1 Q IS :TRUE"
"2 (:-> Q R) IS :TRUE"

(why 1)
"Q IS :TRUE FROM:"
"1 P IS :TRUE"
"2 (:-> P Q) IS :TRUE"

(why 1)
"P IS :TRUE AS A PREMISE"

(why 0)
"Q IS :TRUE FROM:"
"1 P IS :TRUE"
"2 (:-> P Q) IS :TRUE"

(why 0)
"R IS :TRUE FROM:"
"1 Q IS :TRUE"
"2 (:-> Q R) IS :TRUE"

(why 2)
"(:-> Q R) IS :TRUE AS A PREMISE"
```

substitution capabilities of the system.

The final scenario demonstrates the use of simple demons. Of course demons are not normally defined by typing them into the top level RUP environment. Each pattern directed demon has a trigger pattern, a triggering condition keyword (such as **:intern**) and a queue on which the invocation of the body is placed when the triggering occurs. The symbol ***basic-queues*** is bound to a list of queues which are emptied by certain top level functions such as **assert** and **why**. The body of a demon may be any list of LISP expressions. The macro **lconst** constructs clauses in the TMS corresponding to the assertions it is given. Constructing a clause in the TMS is different from asserting an implication; specifically clauses never appear in explanations while asserted implications do.

Scenario 2.

```
(assert '(:-> p r))
 ...
(assert '(:-> q r))
 ...
(assert '(:or p q))
 ...
(why 'r)
"I DON'T KNOW WHETHER OR NOT R IS :TRUE"

(try-to-show 'r)
 ...

(why 'r)
"R IS :TRUE FROM:"
"1 (:-> P R) IS :TRUE"
"2 (:-> Q R) IS :TRUE"
"3 (:OR P Q) IS :TRUE"
```

Scenario 3.

```
(assert '(= (f a b) a))
 ...
(why '(= (f (f a b) b) a))
"(= (F (F A B) B) A) IS :TRUE FROM:"
"1 (= (F A B) A) IS :TRUE"
```

Scenario 4.

```
(setq *user-queue* (make-fifo))
 ...
(setq *basic-queues* (append *basic-queues* (list *user-queue*)))
 ...
(notice (intern (dog ?x)) *user-queue*
   (lconst (:-> (dog ?x) (mammal ?x))))
 ...
(notice (intern (hawk ?x)) *user-queue*
   (lconst (:-> (hawk ?x) (bird ?x))))
 ...
(notice (intern (mammal ?x)) *user-queue*
   (lconst (:not (:and (mammal ?x) (bird ?x)))))
 ...

(assert '(dog fido))
 ...
(why '(hawk fido))
"(HAWK FIDO) IS :FALSE FROM:"
"1 (BIRD FIDO) IS :FALSE"

(why 1)
"(BIRD FIDO) IS :FALSE FROM:"
"1 (MAMMAL FIDO) IS :TRUE"

(why 1)
"(MAMMAL FIDO) IS :TRUE FROM:"
"1 (DOG FIDO) IS :TRUE"

(why 1)
"(DOG FIDO) IS :TRUE AS A PREMISE"
```

3. THE TRUTH MAINTENANCE SYSTEM

A truth maintenance system is a utility which operates on an assertional data base (a collection of TMS nodes) and has at least the following four properties:

1) It can perform some form of propositional deduction from propositional premises (propositional deduction does not involve quantification).

2) It records justifications for deduced assertions and can generate explanations for those assertions.

3) It can incrementally retract deductions when premises are retracted so that all "true" assertions in the data base are either premises or follow logically from the premises.

4) It can perform "dependency directed backtracking". That it to say that when a contradiction arises it can use the recorded justifications to track down the premises underlying that contradiction. Furthermore when one of these premises is retracted it can use the contradiction to deduce the negation of the retracted premise.

This section describes the functionality of RUP's TMS in detail. The first part of this section describes the association between queues and invariants which is used in much of RUP. The second part describes the two basic data structures used in the TMS. The third describes the basic TMS invariants which form the major specifications for the functionality of the TMS. The fourth part describes the major functions defined in the TMS. The fifth part describes TMS demons.

3.1. Queues and Invariants

Much of RUP is specified by stating invariants which should hold in the RUP environment. Several of these invariants are associated with queues, such that for each violation of the invariant there is some entry on the queue that can be used to correct that violation. Thus when a queue has been emptied the invariant associated with that queue must hold. For example there is a TMS invariant which says that for each contradiction in the TMS there is an entry on the queue ***backtracking-invariant***. While there are many TMS invariants, the only user visible queue associated with these invariants is ***backtracking-invariant***. (There are other user visible queues which are associated with other RUP invariants.) The basic primitives for constructing and manipulating all RUP queues are described here.

make-fifo ()

This function returns a new first in first out queue.

fifo-push (item queue)

This function pushes an item on a fifo queue. Each item on a queue must be a list of a function followed by a list of arguments. Thus a particular queue entry **item** can be "run" by evaluating:

(apply (car item) (cdr item))

fifo-empty? (queue)

Thus predicate is non-nil just in case the given queue is not empty.

run-queues (queue-list)

This function takes a list of queues and empties them by "running" the items on the queues. This function iteratively takes the next item of the first non-empty queue in the given list of queues and runs that item. Note that in running one item more items may be queued. Thus a queue which was empty at one iteration may not be empty on the next iteration. On each iteration this function takes the first item off the first non-empty queue. The function terminates when all queues are empty.

The order of the queues in the given list of queues imposes a "priority" on the queues. Items on the second queue will only be run in environments in which the first queue is empty. Thus if there is some invariant associated with the first queue items on the second queue will only be run in an environment in which that invariant is in force.

basic-queues variable

This variable is bound to a list of queues and can be passed as an argument to **multi-fifo-empty**. The default value of this variable is:

(list *equality-invariants* *rup-top-level* *backtracking-invariant*)

The variables ***equality-invariants***, ***rup-top-level***, and ***backtracking-invariant*** are all set to queues in the default RUP environment.

3.2. Nodes and Clauses

There are two basic data structures used in the TMS: **TMS-nodes** and **clauses.**

3.2.1 TMS NODES

TMS-node structure

```
(defstruct (tms-node (:type :named-array))
  assertion
  (truth ':unknown)
  support
  true-noticers
  false-noticers
  change-noticers
  neg-clauses
  pos-clauses
  default
  default-cert
  certainty
  (node-plist (ncons nil))
  (node-extension (funcall *make-node-extension*)))
```

The slots of tms-nodes is described below:

assertion A tms-node is intended to represent a proposition or assertion of some form. The assertion slot is not used by the TMS directly but is intended to hold the name of the assertion. In RUP the assertion slot holds the **term** whose print name is the name of the assertion. Terms are described in the section on the equality system.

truth This slot always contains one of the atoms **:unknown**, **:true** and **:false.**

support This slot is either **nil** or contains a clause which is the justification for the truth of this node. Only nodes whose **truth** is either **:true** or **:false** have non-nil **support** slots. This slot is described in more detail in the section on TMS invariants.

true-noticers, false-noticers, and **change-noticers** These slots contain demons which are queued when certain events occur in the TMS. These slots are described in more detail in the section on tms noticers.

neg-clauses For any TMS node *n* the **neg-clauses** of n is a list of all those clauses *c* such that the pair **(***n* **. :false)** is a member of the **clause-list** of *c.* (See the description of clauses.)

pos-clauses For any TMS node *n* the **pos-clauses** of n is a list of all those clauses *c* such that the pair **(***n* **. :true)** is a member of the **clause-list** of *c.* (See the description of clauses.)

default This slot is only used in the TMS with the semi-automatic premise controller. This slot is either **nil** or contains a default truth value which is either **:true** or **:false**. This slot is described in more detail in the section on TMS invariants.

default-cert This slot is only used in the TMS with the semi-automatic premise controller. If the **default** slot is non-nil then this slot contains an integer between the values of the special variables ***min-cert*** and ***max-cert*** inclusive. This slot is discussed further in the section on TMS invariants.

certainty This slot is only used in the TMS with the semi-automatic premise controller. If the truth of the node is not **:unknown** then this slot contains the minimum certainty of the premises which underly the truth of the node.

node-plist This slot contains a disembodied property list (LISP Machine manual pp. 71-72) and is initialized to **(ncons nil)**. This allows the user to define properties which are not already structure slots. A new "slot" for TMS nodes can be defined as follows:

```
(defmacro new-slot (node)
  '(get (node-plist ,node) 'new-slot))
```

Since the **node-plist** slot is used internally in RUP it is important that the user not violate the conventions for property lists in using this slot.

node-extension This slot is not used internally in RUP and is available for use by the user. The value of this slot is initialized to **(funcall *make-node-extension*)** so that the user can control the initial value. ***make-node-extension*** is initialized to a function which always returns nil. It is intended that the user define his own structure which extends the node data structure and initialized this slot to that structure which could be done as follows:

```
(defstruct (node-extension)
  f1
  f2
  ...)

(defun extension-maker ()
  (make-node-extension))

(setq *make-node-extension* (fsymeval 'extension-maker))
```

3.2.2 CLAUSES

A clause is a data structure which represents a logical disjunction. A clause should be thought of as representing a constraint which says that at least one of a particular collection of items must be true. The details of this data structure are described below.

clause structure

```
(defstruct (clause (:type :named-array))
  clause-list
  psat)
```

clause-list This slot contains a list of pairs such that the **car** of each pair is a tms-node and the **cdr** of each pair is either the atom **:true** or the atom **:false**. A given pair is said to be true if the **truth** of its **car** is the same as its **cdr**. Similarly a pair is said to be false if the **truth** its **car** is the opposite of its **cdr**. For example if the **truth** of a node *n* is **:false** then the pair **(***n* **. :false)** is said to be true while the pair **(***n* **. :true)** is said to be false. Since the **truth** of a TMS node can be **:unknown** it can be the case that a given pair in the clause list is neither true or false. A clause should be thought of as a disjunction which says that not all of the pairs in its **clause-list** can be false.

psat This slot always contains a number which is the number of pairs in the clause list which are not false. Any clause whose **psat** is 0 is called a contradiction. A clause whose **psat** is 1 can be used as a justification for assigning a truth value to the node which is the car of the pair in the **clause-list** which is not false.

3.2.3 SOME CONVENIENT MACROS

The following macros are convenient for testing the **truth** of a node:

true? (node)

The form **(true? node)** macroexpands to **(eq ':true (truth node))**.

false? (node)

(false? node) = => (eq ':false (truth node))

unknown? (node)

(unknown? node) = => (eq ':unknown (truth node))

3.3. The TMS Invariants

There are three groups of invariants concerning the TMS data structures which are maintained by the TMS. The first group of invariants, the justification invariants, ensure that every deductions has a well founded justification (i.e. that the deduction performed by the system is sound). The second group of invariants, the deduction invariants, guarantee that deduction is closed under a simple deduction rule (which is equivalent to unit clause resolution). A final backtracking invariant can be used to ensure that the set of premises in the system is "consistent" in that no contradiction can be deduced using the TMS's deduction machinery. Some of the justification and deduction invariants only apply to the TMS with the semi-automatic premise controller. Only the backtracking invariant involves a user visible queue.

3.3.1 THE JUSTIFICATION INVARIANTS

Only nodes whose **truth** slot is either **:true** or **:false** can have non-null **support** slots and when such a **support** slot is non-null it contains a clause which is the justification for the truth value of the supported node. Each clause should be thought of as a disjunction (see the above description of the clause data structure). The basic idea behind justifications is that truth value of the justified node (the value of its **truth** slot) follows logically from the justifying clause and the truth values of the other nodes in that clause. All clauses are interpreted by the system as logical tautologies, thus while the truth values assigned to nodes can be retracted, clauses cannot be removed. Similarly, in generating explanations the system will list assignments of truth values to nodes but will not mention the existence of clauses. The interpretation of clauses as tautologies is described in more detail in [McAllester 80b]. A TMS node whose **truth** is either **:true** or **:false** (i.e. not **:unknown**) but which does not have any supporting clause (i.e. its **support** slot is **nil**) will be called a *premise.*

Local Support Invariant: This invariant states that the truth value which has been assigned to a supported node *n* follows logically from the clause which is the support of *n* and the truth values which have been assigned to the other nodes appearing in the clause. Specifically let *n* be any TMS node whose **truth** is either **:true** or **:false** and whose **support** is not nil. The **support** of *n* must contain a clause *c* such that the **psat** of *c* is 1 (there is exactly one pair in the clause list of *c* which is not false) and such that the pair in c which is not false contains *n* (the supported node).

Well Founded Support Invariant: This invariant states that support trees are acyclic, i.e. that no node is a support node of itself. Specifically let *n* be any TMS node whose **truth** is either **:true** or **:false** and whose support is some clause *c*. The nodes other than *n* which appear in the clause list of *c* will be called the immediate support nodes of *n* (a premise has no immediate support nodes). A node *m* will be called a support node of *n* if it is either an immediate support node of *n* or it is an immediate support node of some node which is a support node of *n* (thus the support nodes of *n* are those nodes appearing in the support tree of *n*). The premises which are

support nodes of *n* will be called the support premises of *n*.

Certainty Justification Invariants: These are two invariants which apply only to the TMS with the semi-automatic premise controller. To define the first certainty justification invariant let *n* be any premise. The **truth** of *n* must be the same as the **default** of *n* (which must not be nil) and the **certainty** of *n* must be the same as the **default-certainty** of *n* (which must also not be nil). In other words any premise must be a premise by virtue of the fact that it has a default value and the certainty of the premise is the default certainty. To define the second certainty justification invariant let *n* be any node whose **truth** is either **:true** or **:false** and whose **support** is some clause *c*. The **certainty** of *n* must be the minimum of the **certainties** of all of *n*'s immediate support nodes. This together with the well founded support invariant implies that the **certainty** of *n* is the minimum of the **certainties** of all of the support premises of *n*.

3.3.2 THE DEDUCTION INVARIANTS

The TMS performs simple propositional deduction from clauses and the truth values which have been assigned to nodes. The deduction performed is not complete (i.e. there are valid deductions which are not made). However the deduction processing is incremental and is guaranteed to terminate in linear time in the number of clauses in the system. The basic deduction invariant is that all deductions which can be made from a single clause and assignments of truth values to nodes have been made.

Main Deduction Invariant: Let *c* be any clause whose **psat** is 1 (any clause such that there is only one pair in its clause list which is not false). Let *p* be the pair in *c* which is not false and let *n* be the node which is the **car** of *p*. The main deduction invariant is that the **truth** of n is the same as the **cdr** of *p*. If the **truth** of *n* was **:unknown** then the clause *c* could be used to deduce that *n* must be assigned the truth value which is associated with it in *p*. The main deduction invariant says that all such deductions have been made.

Default Value Invariant: This invariant applies only to the TMS with the semi-automatic premise controller. It ensures that any node which has a default truth value and which cannot be proven to have the opposite of its default value does in fact take on its default value. Specifically let *n* be any node whose **default** is not nil (i.e. any node with a default truth value). The default value invariant is that the **truth** of *n* must be either **:true** or **:false** and that the **certainty** of *n* must be at least as large as the **default-cert** of *n* (which must not be nil). Furthermore if the **truth** of *n* equals the **default** of *n* and the **certainty** of *n* equals the **default-cert** of *n* then the **support** of *n* must be **nil** (the node *n* must be a premise).

Deduction Certainty Invariant: This invariant applies only to the TMS with the semi-automatic premise controller. It says that each node is given the strongest (most certain) justification which can be found via the propositional deduction mechanisms used by the TMS. Specifically let *c* be any clause whose **psat** is 1, let *p* be the pair in *c* which is not false, and let *n* be the node which is the **car** of *p*. The deduction certainty invariant is that the **certainty** of *n* is not smaller than the minimum of the certainties of the nodes in the false pairs of *c*. To better understand this invariant consider the relationship between the clause *c* and the node *n*. By the main deduction invariant *n* must be assigned the truth value which is the **cdr** of *p*. However the support of *n* need not be the clause *c*. If *c* is not the support of *n* then *c* may provide an alternative method of deducing the truth value of *n* (the only problem would be if using *c* for the support of *n* would introduce a circular justification violating the well founded support invariant). If the certainty of *n* were less than the certainty which would result from using *c* as the support of *n* then *c* provides a "stronger" argument for the truth value assigned *n* and the support for *n* could be strengthened by setting it to *c* (it can be shown that such "strengthening" never introduces circularities). The deduction certainty invariant says that all such possible strengthenings have been done.

3.3.3 THE BACKTRACKING INVARIANT

Any clause whose **psat** is 0 is called a contradiction. Since each clause is interpreted as a tautological disjunction, if all pairs in a clause *c* are false then the truth values which have been assigned to the nodes in those pairs are mutually contradictory.

The backtracking invariant: This invariant is that for each contradictory clause *c* there is a list *b* on the queue ***backtracking-invariant*** such that the **car** of *b* is the function which is the value of the variable ***backtracker*** and the **cdr** of *b* is a one element list containing *c*. Thus "running" *b* is equivalent to evaluating: **(funcall *backtracker* *c*)**.

backtracker variable

The value of this variable is a backtracking function which is used to construct the item placed on the queue ***backtracking-invariant*** when a contradiction arises. The default value of this variable is **backtracker-default** which is described below.

backtracking-invariant variable

The value of this variable is the queue associated with the backtracking invariant.

3.4. Major TMS Functions

This section describes the TMS functions which might be of interest to the user. It also describes some parameters of the TMS which can be set by the user.

add-clause (clause-list)

The clause-list must be a list of pairs each of which associates a TMS node with either **:true** or **:false**. This function creates a clause with the given clause list and ensures all of the TMS invariants by performing whatever deductions the clause allows and by queueing the backtracking of any resulting contradictions.

node-add-clause (pos-nodes neg-nodes)

The pos-nodes and neg-nodes arguments must both be lists of TMS nodes. This function first constructs a clause list by associating all the nodes in pos-nodes with **:true** and all of the nodes in neg-nodes with **:false**. It then adds a clause with this clause list by calling **add-clause**.

implies (nodes node)

A call to this function of the form **(implies** nodes node**)** is equivalent to **(node-add-clause (list** node**)** nodes**)**. It adds a clause which represents the assertion that if all of the TMS nodes in the nodes argument are true then node should also be true.

contradictory (nodes)

A call to this function is equivalent to **(node-add-clause nil** nodes**)**. It adds a clause which says that one of the nodes must be false.

clause-cert (clause)

This function is only defined in the TMS with the premise controller. This function takes a clause and returns the minimum certainty of the TMS nodes in the false pairs of that clause. The justification certainty invariant says that the certainty of a supported node equals the clause-cert of the support of that node.

make-premise (node truth-value)

This function is only defined in the TMS *without* the premise control mechanism. This function forces the **truth** of the given node to be the given truth value which is required to be either **:true** or **:false**. If the given node was previously assigned the opposite value then

retraction is done before the new assignment is made. This function guarantees that all TMS invariants are maintained.

min-cert*, *max-cert variables

These are variables which may be set by the user. All certainties must be between the values of ***min-cert*** and ***max-cert*** inclusive. The default values of ***min-cert*** and ***max-cert*** are 1 and 5 respectively.

set-default (node value certainty)

This function is defined only in the TMS with the premise control mechanism. This function sets the **default** of the given node to value and the **default-cert** of the given node to the given certainty. This function guarantees that all TMS invariants are maintained by performing whatever truth assignments, retraction, deduction, and backtrack queueing that is necessary. Thus if the **truth** of the given node was unknown before the call then the **truth** of the given node will be set to the given value (which must be **:true** or **:false**).

retract-premise (node)

This function is only defined in the TMS *without* the premise controller. This function sets the **truth** of the given node to **:unknown** and guarantees the maintenance of all TMS invariants. (Maintenance of the invariants requires a retraction phase in which all nodes which depended on the retracted node are retracted and a deduction phase in which all nodes which were retracted are checked to see if some alternative support is available. To avoid circular dependencies it is important that the retraction phase completes before the deduction phase begins. To achieve this there is an internal queue associated with the deduction invariants.)

remove-default (node)

This function is only defined in the TMS with the premise control mechanism. This function sets both the **default** and the **default-cert** of the given node to **nil** and maintains all TMS invariants. Thus the given node will not be a premise after this function exits.

view-node variable

This variable is bound to a function which when applied to a TMS node returns a "name" for that node. The default value for ***view-node*** is the function **view-node-default** which assumes that the **assertion** of the node is a term (described in the section on the equality system) and returns the expression which that term represents.

view-clause (clause)

This function returns an "image" of the clause list of the clause in which each node has been replaced by the "name" of the node as given by the value of ***view-node***.

node-why (node)

This function generates an explanation for the truth value assigned the node. If the **truth** of the given node is **:unknown** then a simple statement to that effect is generated. If **truth** of the given node is **:true** or **:false** and its **support** is not nil then the generated explanation gives a numbered list of the immediately support nodes and their **truths**. The argument to **node-why** may be a number in which case an explanation is generated for the support node corresponding to that number in the previous explanation. If the argument 0 is given then the explanation stack is "popped". The following scenario demonstrates the use of this function.

```
(setq *view-node* '(lambda (node) (assertion node)))
 ...
(defun symbol-node (sym)
  (let ((n (make-tms-node)))
    (set sym n)
    (setf (assertion n) sym)
    sym))
 ...

(symbol-node 'p)
 ...
(symbol-node 'q)
 ...
(symbol-node '|(:-> p q)|)
 ...
(implies (list |(:-> p q)| p) q)
 ...
(make-premise |(:-> p q)| ':true)
 ...
(make-premise p ':true)
 ...

(node-why q)
"q is :true from"
"1 |(:-> p q)| is :true"
"2 p is :true"
t

(node-why 1)
"|(:-> p q)| is :true as a premise"

(node-why 0)
"q is :true from"
"1 |(:-> p q)| is :true"
"2 p is :true"
t

(node-why 2)
"p is :true as a premise"
```

The **node-why** function allows the user to walk around the support tree of a node investigating various support paths.

backtracker-default (clause)

This is the default value of the variable ***backtracker*** which is used in constructing backtracking forms to place on the queue ***backtracking-invariant*** (see the section on the backtracking invariants). This function takes a clause and if the clause is not a contradiction it does nothing. Otherwise it first constructs a list of all the support premises of all the nodes in the clause (all of the premises underlying the contradiction). In the TMS with the semi-automatic premise controller this list is then filtered so that only those premises which have the least certainty remain. One premise from the candidate premises is then chooses for retraction. If there is only one candidate premise then this is the one chosen. If there is more than one candidate premise then the value of ***premise-selector*** is applied to the list of candidate premises and the premise returned is the one chosen for retraction. (The fact that the value of ***premise-selector*** may be called even in the TMS with the premise controller is the reason for calling this premise controller semi-automatic rather than automatic.) The premise chosen for retraction is retracted *and then the negation of that premise is deduced* from the other premises and the fact that the premises are mutually contradictory.

premise-selector variable

The value of this variable must be a function which takes a list of nodes and returns one of them. This function is only called on lists of nodes where the current truth values of the nodes in that list are mutually contradictory. The default value of ***premise-selector*** is **premise-selector-default** which types the list of nodes at the terminal and asks the user to select one.

premises (clause-list)

This function returns a list of all the nodes which are premises which either appear directly in false pairs in the clause-list or are support premises of a node in a false pair in clause-list. This function can be applied to the clause list of a contradiction to get the set of premises underlying the contradiction or it can be applied the clause-list of the support of a node to get the set of premises supporting that node.

reverse-truth (node contradiction)

This function can be used to write backtracking functions. The given contradiction must be a clause whose psat is 0 (a contradiction) and the given node must be a premise underlying

that contradiction. This function forces **truth** of node to be set to the opposite of the value it has when the function is called. All TMS invariants are maintained. Note that in the TMS with the premise controller a call to **reverse-truth** normally results in the other nodes underlying the contradiction being the premises supporting the reversed value of the given node and therefore the certainty of the node ends up being the minimum of the certainty of these premises. However if the given node has a default strength which is greater than the minimum strength of the other premises underlying this contradiction, a problem arises. Specifically the default value invariant says that the certainty of a node with a default value can not be less than the default certainty. If such a problematic reversal is attempted it simply will not "stick" and the system will end up in much the same state that it started in.

node-try-to-show (node value &optional refutation-queues split-nodes (certainty *min-cert*))

This function uses a refutation mechanism to extend the deductive power of the TMS. In the TMS without the premise controller the given node must be a TMS node whose **truth** is **:unknown**. In the TMS with the premise controller either the **truth** of the given node must be **:unknown** or the **certainty** of the node must be less than the given certainty. This function attempts to prove from the premises already in the TMS that the given node must be assigned the given truth value. In the TMS with the premise controller this function attempts to prove that the given node must be assigned the given value using only the premises of of certainty greater than or equal to the given certainty. Thus this function can be used to search for a stronger proof of a truth value assignment which is already in force.

The function **node-try-to-show** works by assuming the negation of the thing to be proven and searching for a contradiction. It takes an optional list of refutation queues which are queues to be emptied after the negation has been assumed. The assumption of the negation may trigger demons which are placed on queues. Running those demons may lead to the deduction of a contradiction based on the assumption which would otherwise not have been found. The function **multi-fifo-empty** is used to empty the queues once the assumption has been made.

The function **node-try-to-show** also takes an optional list of split nodes. If split nodes are provided then an attempt is made to prove that all assignments of truth values to the split nodes imply the desired truth value and therefore that this value holds independent of the truth of the split nodes. This is done by actually assigning all possible combinations of truth values to the split nodes and for each such assignment using the refutation mechanism and the rqueues to try to show that the negation of desired truth value leads to a contradiction.

The following scenario provides an example of the use of this function. The functions used in this scenario are defined elsewhere in this manual. For the following example it is important to note that if there is a constraint in the TMS which says that either *p* or *q* must be true, and *p* is made false, then *q* will be deduced to be true. This is important when *p* is used as split node.

```
(notice (:true (p ?x)) *some-queue*
   (lconst (:-> (p ?x) (r ?x))))
 ...
(notice (:true (q ?x)) *some-queue*
   (lconst (:-> (q ?x) (r ?x))))
 ...
(assert '(:or (p a) (q a)))
 ...
(why '(r a))
"I DON'T KNOW WHETHER OR NOT (R A) IS :TRUE"

(node-try-to-show (virt-tms-node (term '(r a)))
          ':true
          (list *some-queue*)
          (list (virt-tms-node (term '(p a)))))
 ...
(why '(r a))
"(R A) IS :TRUE FROM:"
"1 (:-> (P A) (R A)) IS :TRUE"
"2 (:-> (Q A) (R A)) IS :TRUE"
"3 (:OR (P A) (R A)) IS :TRUE"
```

3.5. TMS Noticers

Each node has three noticer slots, **true-noticers**, **false-noticers**, and **change-noticers**, each of which contains a list of "noticers". A noticer is a cons cell whose **car** is a queue and whose **cdr** contains an item to be placed on that queue when the noticer "triggers". Under certain conditions all of the noticers in a given noticer slot will be triggered and the noticer slot will be set to nil. Thus a given noticer in a given slot will only be triggered once. True noticers (the cells in the **true-noticers** slot) are triggered whenever the node becomes true. False noticers are triggered whenever the node becomes false and change noticers are triggered whenever any change is made either to the **truth** or the **certainty** of the node.

It is often desirable to have a certain demon run whenever a node becomes true rather than just the first time that node becomes true. There is a straightforward way of doing this which is exemplified by the following scenario. When the below function **notice-problem** is applied to a queue and a node it first checks to see if the node is true and if so it applies a special handler to that node. Independent of whether or not the node is true however it places a true noticer on the node using the given queue. This noticer is such that if the node is ever set to true in the future then this function will be called again with the same arguments.

```
(defun notice-problem (queue problem-node)
  (if (eq ':true (truth problem-node))
      (problem-handler problem-node))
  (push (cons queue (list 'notice-problem queue problem-node))
     (true-noticers problem-node)))
 ...
(notice-problem *some-queue* n)
```

4. THE EQUALITY SYSTEM

The equality system is a collection of utilities for handling the substitution of equals for equals. The description of the equality system given here is divided into three parts. The first describes terms and the interning of terms. Terms are analogous to a LISP atoms in that they are interned so that one can guarantee that there are no two distinct terms with the same print name. Unlike LISP atoms however terms can be either atomic or can contain subterms which can be substituted for in the equality system. The second part of this section describes the equality and equivalence class data structures and the equality invariants which specify the substitution computations. The final part of this section describes simplification utilities which allow the user to define a somewhat arbitrary simplicity order on terms and then computes the simplest term which can be equated with any given term via the substitution of equals for equals.

4.1. Terms and Interning

Terms are defined as follows:

```
(defstruct (term (:type :named-array))
  (term-hash (hash-count))
  subterms
  parents
  eqs
  next-canonical
  eq-next-canonical-eqs
  class-data
  term-tms-node
  user-referenced?
  (term-plist (ncons nil))
  (term-extension (funcall *make-term-extension*)))
```

The various slots of this data structure are described below.

term-hash This is an integer which is unique to this term. This integer is used as a hash value for the term.

subterms This slot holds two basically different kinds of information depending on the kind of term involved. If the term is a composite term then this slot holds a list of the subterms of the term (a list of term data structures). For example a term whose print name is **(f a b)** would have subterms whose print names are **f**, **a**, and **b** respectively. If the term is "atomic" then it has no subterms and the **subterms** slot contains the print name of the term. There are three different kinds of atomic terms. First of all there are symbols whose **subterms** slot is simply a LISP symbol. Second there are numbers whose subterm slot is a LISP number. Finally there are quotations whose subterms slot contains a LISP expression whose car is the symbol **quote.** Numbers and quotations are *self-referential* terms. This means that these terms are interpreted as *denoting* themselves. Specifically the term whose print name is the number 1 is taken to

denote the number 1 and the term whose print name is the expression **(quote (f a))** is taken to denote the expression **(f a)**. Self referential terms play an important role in the equality invariants.

parents This slot contains a list of all those terms which contain this term as an immediate subterm (i.e. all those terms which contain this term in their **subterms** slot). This is used in the equality algorithms described in the next part of this section.

eqs This is a list of all the equality data structures which equate this term with some other term. This slot is maintained by the function **make-eq** described elsewhere.

next-canonical This slot is either **nil** or contains a "more canonical" term. The function **e** described below takes a term t and returns its "canonicalization" which is t if the **next-canonical** of t is **nil** and otherwise it is the canonicalization of the **next-canonical** of t. Two terms are equivalent just in case they have the same canonicalization.

eq-next-canonical-eqs The **eq-next-canonical** of a term t is not **nil** just in case the **next-canonical** of t is not **nil** in which case the **eq-next-canonical-eqs** of t contains a list of equality data structures which together imply that t is equal to the **next-canonical** of t.

class-data The **class-data** slot of a term t is not **nil** whenever there is some term s whose **next-canonical** is t. If the **class-data** of a term t is not **nil** then it is a **class-data** data structure which describes the set of terms whose **next-canonical** is t. The **class-data** data structure is described in section 3.2.

term-tms-node This slot is either nil or contains a TMS node representing this term. If a TMS node is present then the term represents an assertion. The function **virt-tms-node** described below takes a term and always returns a TMS node representing that term. For the equality invariants to be maintained it is important that all TMS nodes representing terms be created via **virt-tms-node.**

user-referenced? This slot is a flag which is non-nil just in case this term has been returned as a value of the function **term** or the function **term-hashcons.** This is needed because the equality system creates internal terms via the substitution of equals for equals and it is not desirable to run demons on these terms. Specifically the value of the variable ***new-term*** (described elsewhere) is only applied to terms which are returned from **term** or **term-hashcons.**

term-plist This slot is perfectly analogous to the **node-plist** slot

term-extension This is perfectly analogous to the **node-extension** slot of TMS nodes.

The following functions and variables are relevant to terms and interning.

virt-tms-node (term)

This function returns a TMS node that has been associated with the term. In order to maintain certain TMS-Equality interface invariants it is important that this be the only way in which the **term-tms-node** slot of terms is set.

atomic? (term)

This predicate is true of a term just in case the term is a symbol, a number, or a quotation.

self-referential? (term)

This predicate is true of a term just in case the term is a number or a quotation.

term-tree (term)

This function returns the print name of the term. Curried functions are treated specially (the print form is uncurried) as is described in the section on curried functions.

term (expression)

This is the basic function for interning expressions as terms. The expression argument can either be a number, a symbol, a term, or an arbitrary expression built out of numbers symbols and terms. If the expression is a term then the expression is simply returned. The expression is said to be atomic if it is a number or a symbol or the car of the expression is the symbol **quote**. If the expression is not atomic then this function first recursively computes the list of subterms which is the value of **(mapcar 'term** expression**)**. It then returns the result of applying **term-hashcons** (described below) to this list of subterms. If the expression is atomic then if there is already a term whose print name is the expression then that term is returned. If there is not already such a term then one is created and returned. The value of the variable ***new-term*** is applied to all terms created in this way. This function maintains all of the equality invariants described later.

term-hashcons (subterms)

This function takes a list of subterms and returns a term corresponding to that list of subterms. This function first applies the value of the variable ***intern-canonicalize*** (described below) to the list of subterms. The value of ***intern-canonicalize*** must be a function which

returns either a term or a list of subterms. If a term is returned then that term is simply returned from **term-hashcons**. If a list of subterms is returned then this function looks for an already existing term which has this list of subterms in its **subterms** slot (a hash table is used here for efficiency). If such a term exists it is returned. If no such term exists one is created and returned. The value of the variable ***new-term*** is applied to all terms created in this way. This function maintains all equality invariants.

intern-canonicalize variable

This variable must be bound to a function which takes a list of terms and returns either a term or a list of terms. This is used to map different expressions for the same thing into identical term structures directly in the interning process. For example if a function f is known a-priori to be an a commutative function then the expression **(f a b)** must denote the same thing as the expression **(f b a)**. The default value of ***intern-canonicalize**** is **intern-canonicalize-default** which is described below.

intern-canonicalize-default (subterms)

Terms representing functions of two arguments can be marked as being either associative, commutative, or both (see the macros **associative?** and **commutative?** defined below). For example the term whose print name is **+** is marked as both associative and commutative. The function **intern-canonicalize-default** takes a list of subterms the first of which is a term representing an operator (function or predicate). If the operator term is not marked as being either associative or commutative then the list of subterms is simply returned by **intern-canonicalize-default**. If the operator term is marked as associative then the argument subterms are searched for an application of that same operator and if one is found the arguments in that application are promoted to top level arguments. For example if **f** is marked as an associative operator then **(f a (f b c))** is converted to **(f a b c)**. Since **f** is binary (only binary functions should be marked as associative or commutative) the term **(f a b c)** is interpreted by convention as **(f (f a b) c)**. After the promotion of "internal" arguments for associative operators this function checks to see if the operator is marked as commutative. If so then the arguments are sorted by their **term-hash** slots. Finally the resulting list of subterms (including the operator term) is returned from **intern-canonicalize-default**.

associative? (op-term)

This is a macro which expands as follows:

(associative? op-term) ==> (get (term-plist op-term) 'associative?)

To mark an operator term as associative one simply evaluates:

(setf (associative? op-term) t)

commutative? (op-term)

This macro is just like **associative?**.

new-term variable

The value of this variable must be a function which is applied exactly once to every term which is ever returned from **term** or **term-hashcons**. The default value of this variable is **new-term-default**. For the symbols **=**, **->**, **and**, **or**, etc. to be given the proper interpretations in the top level RUP environment the function **new-term-default** should be called on all terms returned from **term-hashcons**. Thus any function which the user assigns to ***new-term** should call **new-term-default** on its argument if the standard interpretation of the above symbols is desired.

new-term-default (new-term)

This is the default value of the variable ***new-term***. The function **new-term-default** queues applications of hashcons noticers. Hashcons noticers are functions which are associated with an operator term and a queue. When **new-term-default** is applied to a term u it checks the first subterm of u (u's operator) to see if there are any hashcons noticers associated with that operator (if the given new-term is atomic then **new-term-default** does nothing). For each such noticer an application of that noticer to u is placed on the queue associated with the noticer. Noticers are stored on the **hashcons-noticer** property of operator terms. Whenever **new-term-default** is applied to a non-atomic term u the term u is added to the **applications** property of the operator of u.

add-hashcons-noticer (op-term noticer queue)

This function associates the given noticer (which must be a function of one argument) with the given op-term and the given queue. This function queues applications of the given noticer to all currently existing applications of the given op-term.

hashcons-noticers (op-term)

This is a macro which expands as follows:

(hashcons-noticers op-term) ==> (get (term-plist op-term) 'hashcons-noticers)

applications

This is a macro which expands as follows:

(applications op-term) ==> (get (term-plist op-term) 'applications)

4.2. Equalities, Equivalence Classes, and the Equality Invariants

The equality system maintains a congruence relation on terms. The following function can be used to determine whether or not two terms are congruent under this relation.

e (term)

This function is defined as follows:

```
(defun e (term)
  (if (next-canonical term)
      (e (next-canonical term))
      term))
```

Two terms are congruent just in case they have the same image under **e.**

The following defines one of the basic data types in the equality system.

```
(defstruct (equality (:type :named-array))
  term1
  term2
  dependents
  eq-node)
```

The slots of the equality data structure are described below.

term1, term2 These slots contain the terms equated by the equality.

dependents This slot contains the list of all terms which contain the equality in their **eq-next-canonical-eqs** slot.

eq-node This slot holds the tms node which represents the equality.

The equality system is driven by changes in the truth values of the TMS nodes associated with equalities. Therefore the only interesting top level functions for the equality system are for querying the data structures involved.

make-eq (term1 term2 tms-node)

This function should be used uniformly instead of the **make-equality** macro constructed by defstruct. This function creates an equality data structure and sets the **term1**, **term2**, and **eq-node** slots of that structure to the arguments provided. It also updates the **eqs** slot of both terms. Finally it places a change noticer on the given tms-node which will are needed to ensure the equality invariants.

true-eq? (equality)

This macro expands as follows:

(true-eq? equality) ==> (eq ':true (truth (eq-node equality)))

equated-support (term1 term2)

If term1 and term2 are not in the same equivalence class then this function returns nil. If term1 and term2 are in the same equivalence class then this function returns a list of TMS nodes which represent equalities implying the equivalence of term1 and term2.

same-image? (term1 term2)

This predicate is non-nil just in case term1 and term2 have the same number of subterms and those subterms are equivalent in pairs.

class-members (term)

This function returns a list of all terms which are congruent to the given term (all interned terms that is).

equivalents (term)

This function returns a list of terms in the equivalence class of term such that no two terms in that class have the same image (i.e. their subterms are equivalent in pairs). Thus the list of terms returned is the set of "independent" terms equivalent to term.

A term u is said to *point to* a term t just in case either t is the **next-canonical** of u or in case the **next-canonical** of u points to t. A **class-data** data structure c is always contained in the **class-data** slot of exactly one term t called the owner of c. If c is the **class-data** of t then c describes the set of terms which point to t. This data structure is defined as follows:

```
(defstruct (class-data)
  members
  member-referents
  (size 1)
  (class-plist (ncons nil)))
```

The slots of these structures have the following functions:

members The **members** slot of c is a list of terms whose **next-canonical** is the owning term t of c. Note that this is a subset of all the terms which point to t.

member-referents The **member-referents** slot of c contains a list of all self referential terms which point to the owner of c.

size The **size** of c is one plus the number of terms which point to the owner of c.

class-plist This slot is analogous to the **node-plist** slot of TMS nodes and the **term-plist** slot of terms.

The following are the Equality Invariants. These invariants are associated with the queue ***equality-invariants*** and are only guaranteed in environments in which this queue has been emptied.

Equality Justification Invariant: For any term t with a non-null **next-canonical** slot the set of equalities in the **eq-next-canonical-eqs** slot of t are all true equalities (the **truth** of their **eq-nodes** is **:true**) and this set of equalities implies that the u is equal to the **next-canonical** of u.

Congruence Deduction Invariant: For any term t let subterm-image(t) be the term which results from replacing each subterm u of t by e(u) (if t is atomic then subterm-image(t) is just t). The congruence invariant is that for every term t, subterm-image(t) is an interned term which is in the equivalence class of t. This invariant implies that any two terms whose subterms are equivalent in pairs are themselves equivalent. It also implies that any two terms which can be shown equivalent via the substitution of equals for equals are in fact equivalent. For efficiency

reasons this invariant is not maintained on terms which are applications of **=**, **or**, **->**, **and**, **not**, and **iff**. Instead the TMS can be used in conjunction with refutation to show any derivable logical equivalences between these terms.

True Equality Deduction Invariant: The terms of any true equality are both in the same equivalence class.

Derived Equality Deduction Invariant: Let e be any equality such that the terms of e are both in the same equivalence class. There is a clause in the TMS which states that some set of true equalities imply e. Thus if e is **:false** there is a contradiction in the TMS, and if e is not **:false** it must be **:true** (as opposed to unknown).

Assertional Term Invariant: Let t be any assertional term (a term with a **term-tms-node**). If t has a **next-canonical** then the **next-canonical** of t is also an assertional term and there are clauses in the TMS which state that the **eq-next-canonical-eqs** of t imply the equivalence of the **term-tms-node** of t and the **term-tms-node** of the **next-canonical** of t. This invariant ensures that any two assertional terms which are in the same equivalence class are constrained to be logically equivalent.

equate-state variable

This variable is set to a new value each time the congruence relation on terms is changed. This is useful for memoizing computations which depend on the congruence relations. A memoized value is valid as long as ***equate-state*** has the same value that it had when the memoization was done.

4.3. Simplification

The functions described here allow the user to define a simplicity order on terms and then efficiently simplify terms. Specifically let u be some term. The functions described compute a term which is at least as simple under the user defined order as any term which can be equated with u via the substitution of equals for equals based on the premise equalities. For example suppose one has the function symbol + which is to be interpreted as standard addition over the integers and consider a term of the form **(+ (+ x y) z)**. If this term could with a term composed entirely of numerals and the function symbol + then that term could be "evaluated" to yield a numeral equivalent to the original term. The problem of finding an expression for a given term in terms of some subset of "allowed" terms can be solved by defining a simplicity order in which terms containing only allowed symbols are simpler than terms containing symbols which are not allowed.

A simplicity order is defined by setting the following three variables to appropriate functions.

atomic-level variable bound to function with argument list: (atomic-term)

This variable must be bound to a function which takes an atomic term and returns a "level". Levels can be any data structures so long as they are consistent with the values of the next two variables. The default value of this function is **atomic-level-default** described below. The default levels are integers.

subterm-level variable bound to function with argument list: (subterm-levels)

This variable must be bound to a function which takes a list of levels and returns a level which is the level of any term whose subterms have the corresponding levels. The default value of this variable is **subterm-level-default** which simply computes the maximum of the subterm levels.

smaller? (level1 level2)

This variable must be bound to a predicate which takes two levels and returns a non-nil value just in case the first level is "smaller" (i.e. simpler) than the second. The default value of this function is the lisp less than function **<**.

The termination and correctness of the simplification procedures depend on some assumptions about the simplicity order. These assumptions are as follows:

Well Foundedness Assumption: There can be no infinitely decreasing chains of levels.

Monotonicity Assumption: Let s and t be any two terms with the same number of subterms such that s is simpler than t (has a smaller level). There must be some pair of corresponding subterms s' and t' of s and t respectively such that s' is simpler that t'. In other words the function bound to ***subterm-level** must be non-decreasing in each sublevel argument.

Subterm Simplicity Assumption: No term can be simpler than a term it contains as a subterm.

Pseudo Total Order Assumption Let l_1 and l_2 be any two levels such that l_1 is less than l_2. No third level l_3 can be unrelated to both l_1 and l_2 (i.e. l_3 must be either smaller or greater than either l_1 or l_2).

new-simplification-state ()

This function of no arguments must be called each time the user changes the simplification order.

atomic-level-default (atomic-term)

This function is the default value of ***atomic-level***. It takes an atomic term and returns a non-negative integer. If the term has an **atomic-level** property then the value of this property is returned. Otherwise the number returned is 0 for self referential terms and 1000 for all other atomic terms.

atomic-level-prop (term)

This is a macro with the following expansion property:

(atomic-level-prop term) ==> (get (term-plist term) 'atomic-level)

Thus to set the atomic level of a term one simply evaluates:

(setf (atomic-level-prop term) n)

However whenever this is done the function **new-simplification-state** should be called.

sbound (term)

This function takes a term and returns an expression which is the print name of a term (not necessarily an interned term) which is at least as simple as any term which can be equated with the argument via the substitution of equals for equals using the premise equalities.

5. THE TOP LEVEL RUP ENVIRONMENT

The top level RUP environment provides several convenient user level functions such as **assert**, **retract**, and **why** each of which takes expressions and converts them to assertional terms. Demons which trigger on the creation of terms are used to provide automatic interpretations for the logical operators, =, **:->**, **:and**, **:or**, **:not**, and **:iff**. There is also a macro which converts logical constraints represented as a sentence of propositional logic and converts it to an equivalent set of applications of **add-clause**. There are also mechanisms for saving partial RUP environments so that one can return to a known state during debugging.

5.1. Top Level Functions

assert (exp &optional (certainty *max-cert*))

This function first computes the term whose print name is exp and then calls **virt-tms-node** to get a TMS node associated with this term. If the TMS without the premise controller is being used then the function **make-premise** is called on the TMS node and the truth value **:true**. If the TMS with the premise controller is being used then the function **set-default** is called on the node, the truth value **:true** and the the certainty argument to **assert** (note that the default certainty is ***max-cert***). Finally this function applies **multi-fifo-empty** to the value of ***basic-queues***.

retract (exp)

This function first finds the TMS node associated with the term whose print name is exp. It then either calls **retract-premise** or **remove-default** on that node depending on which TMS is being used. This function applies **multi-fifo-empty** to the value of ***basic-queues***.

=-noticer (eq-term) noticer for = on queue ***equality-invariants***

This function is an intern noticer for applications of =. Since eq-term is an application of = the second and third subterms of eq-term are the equated terms. The function **virt-tms-node** is called to get a TMS node representing eq-term and the function **make-equality** is applied to the equated terms and the TMS node.

->noticer (implication-term) noticer for **:->** on queue ***rup-top-level***

This function is an intern noticer for applications of **:->**. This function adds clauses to the TMS which ensure that the symbol **:->** is interpreted as logical implication. For each implication of the form **(:-> p q)** the following clauses are added: (each of the below clauses is written as a list of disjuncts.

```
(((:-> p q) . :false) (p . :false) (q . :true))
((p . :true) ((:-> p q) . :true))
((q . :false) ((:-> p q) . :true))
```

These clauses relate three TMS nodes: the node representing an implication **(:-> p q)**, the node representing the antecedent **p**, and the node representing the consequent **q**. The best way to think about these clauses is that they force the TMS to make all possible deductions concerning these three nodes. For example if the implication and the antecedent are true then the consequent will be deduced via the first of the above clauses. If the antecedent is true and the consequent is false then that same clause is used to deduce that the implication must be false. If the antecedent is false then the second clause can be used to deduce that the implication, and so on.

not-noticer (negation-term) noticer for **not** on queue ***rup-top-level***

This function is an intern noticer for applications of **not**. For each negation of the form **(:not p)** the following clauses are added:

```
(((:not p) . :true) (p . :false))
(((:not p) . :false) (p . :true))
```

or-noticer (disjunction-term) noticer for **or** on queue ***rup-top-level***

This function is an intern noticer for applications of **or**. A disjunction term can have an arbitrary number of disjuncts. For each disjunction of the form **(:or p q ...)** the following clauses are added:

```
(((:or p q ...) . :false) (p . :true) (q . :true) ...)
(((:or p q ...) . :true) (p . :false))
(((:or p q ...) . :true) (q . :false))
 ...
```

and-noticer (conjunction-term) noticer for **and** on queue ***rup-top-level***

For each conjunction of the form **(:and p q ...)** the following clauses are added:

```
(((:and p q ...) . :true) (p . :false) (q . :false) ...)
(((:and p q ...) . :false) (p . :true))
(((:and p q ...) . :false) (q . :true))
 ...
```

iff-noticer (log-eq-term) noticer for **iff** on queue ***rup-top-level***

This function is an intern noticer for applications of **iff**. This function adds clauses to the TMS which constrain the truth of the TMS node associated with the logical equivalence term to be the appropriate function of the truth values of the TMS nodes associated with the equivalenced terms. For each logical equivalence of the form **(:iff p q)** the following clauses are added:

```
(((:iff p q) . :false) (p . :false) (q . :true))
(((:iff p q) . :false) (p . :true) (q . :false))
(((:iff p q) . :true) (p . :true) (q . :true))
(((:iff p q) . :true) (p . :false) (q . :false))
```

why (exp)

If exp is a number then this function simply calls **node-why** on that number. Otherwise this function gets the TMS node associated with the term whose print name is exp, then applies **multi-fifo-empty** to ***basic-queues***, then calls node-why on that node. The following is a typical top level RUP scenario.

```
(assert '(:-> p q))
 ...
(assert '(:-> q r))
 ...
(assert 'p)
 ...
(why 'r)
"R IS :TRUE FROM:"
"1 Q IS :TRUE"
"2 (:-> Q R) IS :TRUE"

(why 1)
"Q IS :TRUE FROM:"
"1 P IS :TRUE"
"2 (:-> P Q) IS :TRUE"

(why 1)
"P IS :TRUE AS A PREMISE"

(why 0)
"Q IS :TRUE FROM:"
"1 P IS :TRUE"
"2 (:-> P Q) IS :TRUE"

(why 0)
"R IS :TRUE FROM:"
"1 Q IS :TRUE"
"2 (:-> Q R) IS :TRUE"

(why 2)
"(:-> Q R) IS :TRUE AS A PREMISE"
```

try-to-show (exp &optional rqueues snodes (certainty *min-cert*))

A call to this function is equivalent to:

(node-try-to-show (virt-tms-node (term exp)) ':true rqueues snodes certainty)

The following scenario uses this function.

```
(assert '(:-> p r))
 ...
(assert '(:-> q r))
 ...
(assert '(:or p q))
 ...
(why 'r)
"I DON'T KNOW WHETHER OR NOT R IS TRUE"

(try-to-show 'r)
 ...

(why 'r)
"R IS :TRUE FROM:"
"1 (:-> P R) IS :TRUE"
"2 (:-> Q R) IS :TRUE"
"3 (:OR P Q) IS :TRUE"
```

what-is (exp)

This function is defined as follows:

```
(defun what-is (exp)
  (sbound (term exp)))
```

why-is (exp)

This function is defined as follows:

```
(defun why-is (exp)
  (why '(= ,exp ,(sbound (term exp)))))
```

termq (exp)

This macro is very much like backquote in LISP. If exp contains no "special" symbols then this macro takes an expression and macroexpand to a form which will evaluate to the term whose print name is that expression. Special symbols are those which start with either "?" or "!". It is assumed that symbols starting with "?" will be bound to terms at eval time and that symbols starting with "!" will be bound to lists of terms. The following are examples of macroexpansions of this form:

```
(termq (f a))        ==> (term '(f a))
(termq (f ?a))       ==> (term-hashcons (list (term 'f) ?a))
(termq (g . !args))  ==> (term-hashcons (cons (term 'g) !args))
```

lconst (exp)

This is a very useful macro for adding clauses in the TMS. This macro takes a logical expression and macroexpands to a set of add-clauses representing that expression. This macro treats symbols starting with "?" or "!" in much the same as does **termq**. The following is a list of sample macroexpansions:

```
(lconst (:-> (:and p1 p2 p3) r))

==>(add-clause (list (cons (virt-tms-node (term 'p1)) ':false)
                   (cons (virt-tms-node (term 'p2)) ':false)
                   (cons (virt-tms-node (term 'p3)) ':false)
                   (cons (virt-tms-node (term 'r)) ':true)))

(lconst (:-> (:and (forall (x) (:-> (p x) (q x)))
                   (p ?a))
             (q ?a)))

==>(add-clause (list (cons (virt-tms-node
                             (term '(forall (x) (:-> (p x) (q x)))))
                           ':false)
                   (cons (virt-tms-node
                             (term-hashcons (list (term 'p) ?a)))
                           ':false)
                   (cons (virt-tms-node
                             (term-hashcons (list (term 'q) ?a)))
                           ':true)))

(lconst (:iff p q))

==>(progn
     (add-clause (list (cons (virt-tms-node (term 'p)) ':false)
                       (cons (virt-tms-node (term 'q)) ':true)))
     (add-clause (list (cons (virt-tms-node (term 'p)) ':true)
                       (cons (virt-tms-node (term 'q)) ':false))))
```

Note that **(lconst (:-> p q))** is different from **(assert (:-> p q))** in that the former does not create a term or a tms node representing **(:-> p q)** but instead simply installs a clause in the TMS, while the latter creates a term and a TMS node representing **(:-> p q)** and then asserts that that TMS node is true.

assertq, retractq, whyq, try-to-showq, what-isq, why-isq

These macros are just like **assert**, **retract**, etc. except that they use **termq** to quote there arguments. Thus **(assertq p)** is just like **(assert (termq p))**.

5.2. Initialization

When trying to debug code which interacts with the utilities in RUP it is easy to become confused about the current state of the RUP environment. It would be nice to be able to save the state of the RUP environment at some point and be able to return to that state at some latter point. This section describes some features of RUP which approximate this behavior.

term-init ()

This function of no arguments flushes all existing terms so that any term which is subsequently returned from either the function **term** or the function **term-hashcons** is a completely new data structure. This has serious ramifications for the RUP environment. It means that there are no longer any noticers attached to any accessible operator terms (since those terms are new structures and have no noticers attached). It means that the simplification properties attached to terms have been effectively flushed. It means that the commutative and associative properties of operator terms have been flushed.

perm-init-forms variable

This variable is bound to a list of LISP expressions which get evaluated when RUP is initialized and thus the forms on this list determine the state of RUP which results from an initialization. This is the mechanism provided by RUP for "saving" or "defining" RUP environments. The default value of ***perm-init-forms*** is a list of forms which restore the default RUP environment. The forms in ***perm-init-forms*** get evaluated in the *reverse* of the order in which they appear. Thus the last thing pushed onto the list is the last thing evaluated during initialization. It is important that any forms which change the intern canonicalization process are evaluated before the interning of any term affected by that change. For example it is important that the term for = be marked as commutative before any applications of that term are interned.

temp-init-forms variable

This variable is just like ***perm-init-forms*** except that its default value is nil. The intended use of this variable is described in the below documentation of the functions **fix-temps** and **rup-init**.

fix-temps ()

This function of no arguments is defined as follows:

```
(defun fix-temps ()
  (setq *perm-init-forms* (append *temp-init-forms* *perm-init-forms*))
  (setq *temp-init-forms* nil))
```

The basic philosophy behind this function is that as one develops a RUP environment one can push forms onto ***temp-init-forms*** which will to some extent recreate the environment being developed. Then when one wishes to store that environment so that it will be reconstructed after an initialization one calls the function **fix-temps.**

rup-init (&optional save-flag)

This function calls **term-init**, and evaluates the forms in ***perm-init-forms*** in the reverse of the order in which they appear on the list (i.e. the forms are evaluated in the order in which they were placed on the list). Finally if the save-flag argument is not nil it evaluates the forms on ***temp-init-forms*** in reverse order. If the save flag is nil then it sets ***temp-init-forms*** to nil.

6. THE NOTICE MACRO

This section describes a macro which is used to define demons which trigger on certain events in the RUP environment.

notice ((event pattern) queue &rest body-forms)

The notice macro defines demons which are queued when certain events take place in the RUP environment. The event argument must be one of several meaningful keywords and the pattern argument is an expression which may contain "variables" which are symbols starting with either "?" or "!". The queue argument must be a form which evaluates to a queue and the body-forms can be any lisp expressions to be evaluated when the demon runs (i.e. they are the body of the demon). The details of the **notice** macro are best described through examples. Initially only the keyword **:intern** will be considered.

6.1. Creating Intern Noticers

When an application of **notice** is macroexpanded two function definitions are created by side effect and the **notice** form macroexpands to an application of add-hashcons-noticer. The function definitions must be explicitly evaluated using the macro **include-end-forms**. Consider the following example:

```
(notice (:intern (p ?a)) *user-queue*
    (lconst (-> (p ?a) (q ?a))))

(include-end-forms)
```

This macroexpands to:

```
(progn (add-hashcons-noticer (term 'p) '|(P ?A)-UNIFIER| *user-queue*)
       (push '(add-hashcons-noticer (term 'p) '|(P ?A)-UNIFIER| *user-queue*)
          *temp-init-forms*))


(progn 'compile

  (defun |(P ?A)-UNIFIER| (term)
    (let ((args (cdr (subterms term))))
      (if args
       (let ((?a (car args)))
         (if (null (cdr args))
          (|(P ?A)-BODY| ?a))))))

  (defun |(P ?A)-BODY| (?a)
    (add-clause (list (cons (virt-tms-node
                      (term-hashcons (list (term 'p) ?a)))
                    ':false)
                (cons (virt-tms-node
                    (term-hashcons (list (term 'q) ?a)))
                  ':true)))))
```

In the above expansion the **notice** form macroexpands into a progn which both installs a symbol as a noticer and pushes a form onto ***temp-init-forms*** (***temp-init-forms*** can be used to re-create a RUP environment during initialization as is described elsewhere). Because the demons created by **notice** are implemented as intern noticers associated with operator terms it is important that the car of the pattern not contain variables to be bound during the triggering process. The form **(include-end-forms)** macroexpands into a list of function definitions. The first function defined in the above example takes the term and performs the unification of the term and the pattern. The second function takes the bindings derived from this unification and executes the body of the noticer. The need for two functions (as opposed to a single function which does both unification and executes the body) involves keywords other than **:intern**. The need for **include-end-forms** should be clear from the following more complex example involving embedded demons.

```
(notice (:intern (function-from ?f ?domain ?range)) *user-queue*
   (notice (:intern (?f ?x)) *user-queue*
      (lconst (-> (and (function-from ?f ?domain ?range)
                       (?domain ?x))
                  (?range (?f ?x))))))

(include-end-forms)
```

The above macroexpands to:

```
(progn (add-hashcons-noticer (term 'function-from)
                             '|(FUNCTION-FROM ?F ?DOMAIN ?RANGE)-UNIFIER|
                             *user-queue*)
       (push '(add-hashcons-noticer (term 'function-from)
                                    '|(FUNCTION-FROM ?F ?DOMAIN ?RANGE)-UNIFIER|
                                    *user-queue*)
             *temp-init-forms*))

(progn 'compile
 (defun |(FUNCTION-FROM ?F ?DOMAIN ?RANGE)-UNIFIER| (term)
   (let ((args (cdr (subterms term))))
     (if args
      (let ((?f (car args)))
        (if (cdr args)
            (let ((?domain (cadr args)))
          (if (cddr args)
              (let ((?range (caddr args)))
                (if (null (cdddr args))
                 (|(FUNCTION-FROM ?F ?DOMAIN ?RANGE)-BODY|
                    ?f ?domain ?range)))))))))

 (defun |(FUNCTION-FROM ?F ?DOMAIN ?RANGE)-BODY| (?f ?domain ?range)
   (add-hashcons-noticer ?f
                '(lambda (term)
                   (|(?F ?X)-UNIFIER| term ',?f ',?domain ',?range))
                *user-queue*))

 (defun |(?F ?X)-UNIFIER| (term ?f ?domain ?range)
   (let ((args (cdr (subterms term))))
     (if args
      (let ((?x (car args)))
           (if (null (cdr args))
               (|(?F ?X)-BODY| ?x ?f ?domain ?range))))))
```

```
(defun |(?F ?X)-BODY| (?x ?f ?domain ?range)
  (add-clause (list (cons (virt-tms-node
                           (term-hashcons
                            (list (term 'function-from)
                               ?f ?domain ?range)))
                          ':false)
                    (cons (virt-tms-node
                           (term-hashcons (list ?domain ?x)))
                          ':false)
                    (cons (virt-tms-node
                           (term-hashcons
                            (list ?range
                               (term-hashcons (list ?f ?x)))))
                          ':true))))
```

The embedding of the **notice** macro in the above example is very similar to embedding of PLANNER or AMORD demons. The variables in the inner demon inherit their bindings from the outer demon. In any use of **notice** the car of the pattern must not contain variables to be bound during triggering. However the car of the pattern may contain variables which are bound outside the **notice** construct. Note that the internal **notice** form macroexpands to an application of **add-hashcons-noticer** involving the function |(?F ?X)| (the variable ***temp-init-forms*** is not effected by the inner noticer). Without the macro **include-end-forms** it would be very hard for the internal **notice** form to define functions in such a way that they could be compiled.

6.2. Naming Conventions for Noticer Functions

Two functions are defined by side effect each time an application of **notice** is macroexpanded. Each function is given a name which is a symbol interned in the RUP package (or simply an interned symbol in MACLISP). The names of the functions are derived from the pattern in the **notice** form (as shown in the above examples). However special care has been taken to allow for more than one demon with the same pattern. For example the following

```
(notice (:intern (p ?x)) ...)

(notice (:intern (p ?x)) ...)

(notice (:intern (p ?x)) ...)

(include-end-forms)
```

macroexpand to:

```
(progn (add-hashcons-noticer (term 'p) '|(P ?X)-UNIFIER| ...)
       (push '(add-hashcons-noticer ...)
          *temp-init-forms*))

(progn (add-hashcons-noticer (term 'p) '|(P ?X)-2-UNIFIER| ...)
       (push '(add-hashcons-noticer ...)
          *temp-init-forms*))

(progn (add-hashcons-noticer (term 'p) '|(P ?X)-3-UNIFIER| ...)
       (push '(add-hashcons-noticer ...)
          *temp-init-forms*))
```

```
(progn 'compile
 (defun |(P ?X)-UNIFIER| (term)
   ...)
 (defun |(P ?X)-BODY| (?x)
   ...)
 (defun |(P ?X)-2-UNIFIER| (term)
   ...)
 (defun |(P ?X)-2-BODY| (?x)
   ...)
 (defun |(P ?X)-3-UNIFIER| (term)
   ...)
 (defun |(P ?X)-3-BODY| (?x)
   ...))
```

In spite of the function naming convention exemplified above naming conflicts can occur when two demon defining files share a trigger pattern and at least one of the files is compiled. Specifically when a compiled file is loaded the names of the functions defined by that file are the names given at compile time rather the names which would have been generated had the demon definitions been macroexpanded at load time. Consider a compiled file containing a definition for the function |(P ?X)-BODY|. If such a file is loaded into a RUP environment which already has a definition for |(P ?X)-BODY| a naming conflict will occur. It is also important to note that since loading a compiled file does not induce macro expansions it also does not effect the names generated by later macro expansions. The best policy is to make sure that no two files share noticer patterns.

6.3. Events

There are several meaningful event keywords other than **:intern**. These event keywords are described below.

:true

Demons defined using this keyword are triggered whenever the TMS node associated with a term matching the given pattern becomes true. The following example demonstrates the use of this function.

```
(notice (:true (p ?x)) *user-queue*
   (1const (-> (p ?x) (q ?x))))

(include-end-forms)
```

This macroexpands to:

```
(progn (add-hashcons-noticer (term 'p) '|(P ?X)-UNIFIER| *user-queue*)
       (push '(add-hashcons-noticer (term 'p) '|(P ?X)-UNIFIER| *user-queue*)
          *temp-init-forms*))

(progn 'compile
  (defun |(P ?X)-UNIFIER| (term)
    (let ((args (cdr subterms)))
      (if args
      (let ((?x (car args)))
        (if (null (cdr args))
            (|(P ?X)-BODY| term ?x))))))

  (defun |(P ?X)-BODY| (term ?x)
    (if (not (eq ':true (truth (virt-tms-node term))))
     (push (cons *user-queue*
                 '(|(P ?X)-BODY| ,term ,?x))
           (true-noticers (virt-tms-node term)))
     (add-clause (list (cons (virt-tms-node
                              (term-hashcons (list (term 'p) ?x)))
                             ':false)
                       (cons (virt-tms-node
                              (term-hashcons (list (term 'q) ?x)))
                             ':true))))))
```

Note that the code for |(P ?X)-BODY| first checks to see if the tms node associated with triggering term is true. If it is not then a call to |(P ?X)-BODY| is placed on the true-noticers of the node associated with the triggering term. Note that since a node can become true and then unknown before its true-noticers are run |(P ?X)-BODY| might be run several times before it is run in an environment in which the node associated with the triggering term is true.

:false

This keyword is just like **:true** except that the demon is queued when the node associated with the triggering term becomes false rather than true.

:change

This keyword causes the demon to be queued the first time the **truth** of the node associated with the triggering term changes.

:whenever-true

This keyword causes the body of the demon to be run whenever the node associated with the triggering term becomes true. For example the following

```
(notice (:whenever-true (trouble ?x)) *user-queue*
   (trouble-fixer ?x))

(include-end-forms)
```

Gives rise to the following definition:

```
(defun |(TROUBLE ?X)-BODY| (term ?x)
  (push (cons *user-queue*
              '(|(TROUBLE ?X)-BODY| ,term ,?x))
     (true-noticers (virt-tms-node term)))
  (if (eq ':true (truth (virt-tms-node term)))
      (trouble-fixer ?x)))
```

:whenever-false

This is the dual of **:whenever-true.**

:whenever-change

This causes the body of the demon to be run every time the tms node associated with the triggering term changes its truth state.

6.4. List Variables

It is often desirable to be able to write demons which trigger on terms with an arbitrary number of top level arguments. A mechanism for doing this exists and is exemplified by the following definition of a noticer for **list.**

```
(notice (:intern (list . !args)) *user-queue*
   (let ((?first (car !args)))
     (lconst (-> list-definition
                 (= (first (list . !args)) ?first))))
   (if (cdr !args)
       (let ((!rest (cdr !args)))
         (lconst (-> list-definition
                     (= (tail (list . !args)) (list . !rest)))))))
```

A Symbol starting with "!" is interpreted as a variable in a noticer trigger pattern and differs from a symbol starting with "?" only in that it is bound to a list of terms rather than a single term. An error is triggered if either "?" or "!" variables are used in a syntactically incorrect manner.

6.5. Some Useful Macros

This section describes some macros which can be used in conjunction with **notice.**

nlet

The macro **nlet** is just like the macro **let** except that it expands **notice** forms which appear in its body and allows those notice forms to inherit variables bound by the **nlet. notice** forms can only inherit variables bound by surrounding **notice** and **nlet** contexts. The following is an example of the use of **nlet:**

```
(notice (:intern (f . !args)) *user-queue*
   (nlet ((?first (car !args)))
      (notice (:true (r ?first ?other-thing)) *user-queue*
     (let ((!other-args (cons ?other-thing (cdr !args))))
       (!const (-> (r ?first ?other-thing)
                (= (f . !args) (f . !other-args))))))))
```

self

This macro of no arguments is used inside the body of a **notice** form. An application of **self** macroexpands to a form which evaluates to a form which can be placed on a queue and is in fact the current invocation of the body of the innermost demon. Consider the following example:

```
(notice (:true (p ?x)) *user-queue*
   (nlet ((n1 (virt-tms-node (termq (p ?x)))))
      (notice (:true (q ?x)) *user-queue*
      (if (not (true? n1))
          (push (cons *user-queue* (self))
             (true-noticers n1))
          (let ((n2 virt-tms-node (termq (p ?x))))
            (if (not (true? n2))
             (push (cons *user-queue* (self))
                (true-noticers n2))
             (print '((p ,(term-tree ?x))
                   and '(q ,(term-tree ?x))
                   are both true))))))))
```

Note that the print statement will only be reached in an RUP environment where both the nodes associated with the triggering terms are true. If the body of the inner noticer is run in an environment where the node associated with the first triggering term is false (which can happen) then an execution of the body is requeued. The macro **self** creates a new invocation of the innermost **notice** body with the current binding environment. During subsequent invocations of this body either node may be false and the body continues to requeue itself until it is invoked when both nodes are true.

this-noticer

The macro **this-noticer** of no arguments macroexpands to a form which evaluates to the intern noticer placed on an operator term by the innermost **notice** form containing this macro. This allows one to get access to the noticer and remove it once it has fired. Consider the following example:

```
(notice (:true (p ?x)) *user-queue*
   (notice (:true (r ?x ?y)) *user-queue*
      (setf (intern-noticers (term 'r))
         (delete (this-noticer) (intern-noticers (term 'r))))
      ...))
```

The above code might be used when it is known that for any ?x there is at most one ?y such that (r ?x ?y). Thus when a term triggers the inner demon the intern noticer placed on r can be removed thus saving a unification attempts each time some new application of r is interned.

There are cleaner ways to gain efficiency than removing noticers. The section on currying is important for anyone worried about efficiency in demonic triggering.

mapfetch ((var pattern) &rest body-forms)

This macro allows one to access exactly those currently interned terms which match a given pattern. For each such term the body forms are evaluated sequentially in ar environment in which the variable var is bound to the matching term and all of the variables in the pattern are bound to the terms resulting from the match. **mapfetch** returns a list of the values given by the last body form. The fact that the pattern is known at macroexpansion time allows the unification process to be open coded as it is in the functions created by **notice**. Consider the following example:

```
(mapfetch (uterm (p ?x (f ?y)))
       (cons uterm
          (list (cons '?x ?x) (cons '?y ?y))))
```

This evaluates to a list of pairs each of which is a pair of a term and a binding list where each binding list is a list of pairs of a variable and its associated value. **mapfetch** can inherit variable bindings from surrounding **notice** and **nlet** forms as is shown in the following example.

```
(notice (:true (p ?x)) *user-queue*
   (putprop (term-plist ?x)
         (mapfetch (uterm (r ?x ?y))
               ?y)
         'r-relations))
```

The body function defined by this noticer would be as follows:

```
(defun |(P ?X)-BODY| (?x)
  (putprop (term-plist ?x)
        (del-if 'null
             (mapcar '(lambda (uterm)
                     (let ((args (cdr (subterms uterm))))
                    (if args
                        (if (eq ?x (car args))
                         (if (cdr args)
                             (let ((?y (cadr args)))
                               (if (null (cddr args))
                                ?y)))))))
                  (applications (term 'r))))
        'r-relations))
```

An alternative to the above is:

```
(defmacro r-relations (term)
  (get (term-plist term) 'r-relations))

(notice (:true (p ?x)) *r-queue*
   (notice (:whenever-change (r ?x ?y)) *r-queue*
      (if (true? (virt-tms-node (termq (r ?x ?y))))
       (if (not (memq ?y (r-relations ?x)))
           (push ?y (r-relations ?x)))
       (setf (r-relations ?x)
          (delete ?y (r-relations ?x))))))
```

The above code ensures that if ***r-queue*** is empty the for each ?x such that (p ?x) is true (r-relations ?x) is a list of exactly those terms ?y such that (r ?x ?y) is true.

6.6. Currying

This section describes a technique for writing more efficient demons. The basic idea is that when one has a trigger pattern of the form (p t1 ?x t2) where ?x is a variable and t1 and t2 are known terms one can replace that trigger pattern by a pattern of the form (op ?x) where op is a known term incorporating p, t1, and t2. In this way the unification function is not applied to all applications of p but is instead only applied to a select set of terms which contain the known subterms t1 and t2.

There are some conventions adopted in RUP for making this type of transformation more convenient. Specifically there is a special higher order operator called **curry** which takes any number of arguments the first of which is always an operator and the remainder of which are either the number 1 or the number 2. Each of the numeric arguments to **curry** corresponds to an argument of the operator argument to **curry**. The best way to describe **curry** is with some examples. For any binary operator ?r, three place operator ?f, and terms ?x ?y and ?z we have the following equivalences:

```
(?r ?x ?y) = (((curry ?r 1 2) ?x) ?y)
           = (((curry ?r 2 1) ?y) ?x)

(?f ?x ?y ?z) = (((curry ?f 1 1 2) ?x ?y) ?z)
              = (((curry ?f 1 2 1) ?x ?z) ?y)
              = (((curry ?f 2 1 1) ?y ?z) ?x)
              = (((curry ?f 1 2 2) ?x) ?y ?z)
              = (((curry ?f 2 1 2) ?y) ?x ?z)
              = (((curry ?f 2 2 1) ?z) ?x ?y)
```

The above equivalences are enforced by a collection of demons which could have been defined using **notice** as follows:

```
(notice (:intern (curry ?f 2 1 2)) *rup-top-level*
   (notice (:intern (?f ?x ?y ?z)) *rup-top-level*
      (lconst (= (?f ?x ?y ?z)
           (((curry ?f 2 1 2) ?y) ?x ?z)))))
```

These demons are only triggered when **curry** is used so there is no overhead for users who do not use currying. However if currying is ever used in writing efficient noticers the above demons ensure that the correctly curried versions of the appropriate assertions are always created. The **curry** demons are hand coded

for maximal efficiency.

The following example illustrates the use of currying for efficiency.

```
(notice (:true (transitive ?r)) *rup-top-level*
   (notice (:intern (?r ?x ?y)) *rup-top-level*
      (notice (:intern (((curry ?r 1 2) ?y) ?z)) *rup-top-level*
      (!const (-> (and (transitive ?r)
                       (?r ?x ?y)
                       (((curry ?r 1 2) ?y) ?z))
                  (((curry ?r 1 2) ?x) ?z))))))
```

Note that while all uncurried forms are equated with their curried equivalents the curried forms are not necessarily equated with their uncurried equivalents. Thus interning the term **(r a b)** will trigger an intern demon whose pattern is **(((curry r 1 2) ?x) ?y)** but interning the term **(((curry r 1 2) ?x) ?y)** will not trigger an intern demon whose trigger pattern is **(r ?x ?y)**. This fact can be important to writing efficient demons (and is in fact important in the above example).

The function **term-tree** recognizes curried forms and uncurries them which makes them much more readable.

6.7. Redundancy and Completeness

There are some problems with the pattern directed demonic invocation mechanisms described in this section. These problems relate both to the redundant triggering of demons (triggering a demon more often than need be) and to the completeness of triggering (not triggering demons when they should be triggered). Consider the following demon for **pair**.

```
(notice (:intern (pair ?a (list . !rest))) *user-queue*
   (!const (-> list-definition
               (= (pair ?a (list . !rest))
               (list ?a . !rest)))))
```

Suppose the term **(pair a (list b c))** has been interned and that the above demon has been triggered on this term. Further suppose that the equality **(= a 'nil)** is true. Some process may create the term **(pair 'nil (list b c))** as the result of substituting **'nil** for **a** in **(pair a (list b c))**. If the above demon has been triggered on **(pair a (list b c))** then there is no reason to trigger it on **(pair 'nil (list b c))** since these two terms can be equated by substitution. However when the latter term is interned the above intern demon would be triggered.

The result of matching a demon pattern against a particular term is a binding environment e which maps the variables in the pattern to terms. In general two binding environments e_1 and e_2 will be called *variants* of each other if they are defined on the same domain of variables and for each variable ?x in that domain e_1(?x) and e_2(?x) are in the same RUP equivalence class. In general a specific invocation of a demon under a binding environment e will be called redundant if that demon has already been run under a binding environment which is a variant of e. RUP attempts to avoid executing redundant demon invocations by not triggering demons with terms which are generated internally via the substitution of equals for equals. Unfortunately there are cases in which it is useful to run redundant invocations of a demon. For example consider the

following:

```
(notice (:intern (cons ?a ?b)) *user-queue*
   (if (and (eq 'quote (car (subterms ?a)))
            (eq 'quote (car (subterms ?b))))
       (let ((?qterm (term '(quote ,(cons (cadr (subterms ?a))
                                          (cadr (subterms ?b)))))))
          (lconst (-> cons-definition
                      (= (cons ?a ?b) ?qterm))))))
```

Clearly the demon has an important effect when run under a binding environment e which binds the variables to quotations even if the demon has previously been run on a variant of e which did not bind the variables to quotations. The reason the redundant invocation is useful in this case is that the body of the demon tests for syntactic properties of the terms to which the variables are bound. If the body of a demon only uses variables in "semantic" ways then this problem would not arise. A variable is used in a semantic way when it does not matter what term the variable is bound to as long as that term refers to the proper thing.

One possible extension to the existing demonic mechanisms which might solve the problems related to redundant triggering is to introduce a new kind of variable into the patterns of demons which would only bind to self-referential terms. This would allow the syntactic tests made in the above demon to be incorporated into the pattern match and thus one might be able to automatically control demonic invocation in a way that avoids redundant invocations yet still invokes syntactic demons with the proper binding environments.

In addition to having problems with redundant invocations RUP has a problem in that the demonic invocation mechanism is not complete. Consider the following demon:

```
(notice (:intern (f (g ?x))) *user-queue*
   (lconst (-> f-g-definitions
               (= (f (g ?x)) ?x))))
```

Suppose that the term **(f b)** has been interned and that **b** and **(g c)** are in the same equivalence class. By substitution it would be possible to generate the term **(f (g c))** and the above demon could trigger on this term. However since such substitutions are not performed automatically the above demon would not be triggered in this case.

For any expression p containing variables (i.e. any trigger pattern) and any substitution e for the variables in p let e(p) denote the result of replacing each variable in p by its image under e. Let T be any collection of terms and p be any trigger pattern. A substitution e will be said to map p into T just in case e(p) is equivalent (can be equated via substitution of equals for equals) to some term in T. Let $\{p_i\}$ be a collection of trigger patterns each of which is associated with a body b_i. A particular demonic invocation mechanism will be said to be complete with respect to $\{p_i\}$ and T just in case for every p_i and every binding context e which maps p_i into T the body b_i gets called under some binding context which is a variant of e.

It should be possible to extend the demonic invocation mechanism in RUP so that it is complete with respect to the intern demons and the interned terms, the true demons and the true terms, etc. If the demonic invocation mechanism were also careful not to perform redundant invocations such an extension to completeness would probably not generate an unreasonable number of demon invocations.

The problem of generating a complete unification mechanism has been studied in detail by people working on resolution theorem proving. The problem is defined precisely by Huet and Oppen in a survey of results on equations and rewrite rules [Huet & Oppen 79].

6.8. Transitive Relations

There are true noticers defined in the default RUP environment which recognize applications of the second order predicates **transitive**, **reflexive**, **antisymmetric** and **strictly-antisymmetric**. Assuming that the queues ***equality-invariants***, ***rup-top-level***, and ***backtracking-invariant*** have all been emptied the following conditions hold with regard to these predicates:

(1) If an assertional term of the form **(transitive r)** is true then all applications of r which can be deduced from transitivity and known applications of r have been deduced.

(2) If an assertion of the form **(reflexive r)** is true then for each interned term of the form **(r x y)** if x and y are in the same equivalence class then **(r x y)** is true.

(3) If an assertion of the form **(antisymmetric r)** is true then for each pair of true assertions **(r x y)** and **(r y x)** the assertion **(= x y)** is true.

(4) If an assertion of the form **(strictly-antisymmetric r)** is true then for each true assertion **(r x y)** the assertion **(r y x)** is false.

7. FUNCTION AND VARIABLE INDEX

8. BIBLIOGRAPHY

[de Kleer & Sussman 78] de Kleer, J., and Sussman, G. J.
Propagation of Constraints Applied to Circuit Synthesis
MIT AI Lab Memo 485 (September 1978).

[Downey et. al. 80] Downey, P. J., Sethi, R, Tarjan, R. E.
Variations on the Common Subexpression Problem.
J. ACM 27, 4 (Oct. 1980) 758-771.

[Doyle 78] Doyle, J.
Truth Maintenance Systems for Problem Solving
MIT AI Lab Technical Report 419 (September 1978)

[McAllester 80a] McAllester, D. A.
The Use of Equality in Deduction and Knowledge Representation
MIT AI Lab Technical Report 520 (February 1980)

[McAllester 80b] McAllester D. A.
An Outlook on Truth Maintenance
MIT AI Lab Memo 551 (August 1980)

[McAllester 81] McAllester, D. A.
Solving Uninterpreted Equations
MIT AI Lab Working Paper ?? (September 1981)

[McCarthy 80] McCarthy, J.
Circumscription - a form of Non-Monotonic Reasoning
Artificial Intelligence, 13 (1, 2) (April 1980) 27-40

[McDermott & Doyle 80] McDermott, D. and Doyle, J.
Non-monotonic Logic I
Artificial Intelligence, 13 (1, 2) (April 1980) 81-132

[Nelson & Oppen 80] Nelson, G., Oppen, D. C.
Fast Decision Procedures based on Congruence Closures.
J. ACM 27, 2 (April 1980), 356-364.

[Reiter 80] Reiter, R.
A Logic for Default Reasoning
Artificial Intelligence, 13 (1, 2) (April 1980) 81-132

[Stallman & Sussman 77] Stallman, R. M. and Sussman, G. J.
Forward Reasoning and Dependency Directed Backtracking in a system for Computer-Aided Circuit Analysis
Artificial Intelligence, 9 (1977), 135-196.