UM Nonlin Version 1.2.2 User Manual

Subrata Ghosh, James Hendler, Subbarao Kambhampati, Brian Kettler

Parallel Understanding Systems Group
Department of Computer Science
University of Maryland
College Park, MD 20742
Email: nonlin-users-request@cs.umd.edu

Contents

1 Introduction

This user manual describes a UM Nonlin, a Common Lisp implementation of Nonlin. Nonlin is a hierarchical, nonlinear, domain-independent planning system originally developed by Austin Tate. Full details can be obtained from Tate's paper, *Project Planning Using a Hierarchic Nonlinear Planner*, Research Report No. 25, Department of Artificial Intelligence, University of Edinburgh. The implementation of UM Nonlin is relatively faithful to the methodology of the original Nonlin, although some name changes have been made within the operators to conform more closely to more recent usage in the artificial intelligence planning community. The system has been tested in several Common Lisp implementations including Macintosh Common Lisp (MCL) and Allegro Common Lisp, but of course we cannot guarantee that it will work in all common lisp implementations.

The purposes of this user manual are to describe how to install and run the UM Nonlin program, henceforth referred to as "Nonlin" (versus "Tate's Nonlin"), and to highlight the differences between Tate's formulation and this implementation. Thus the algorithms of Nonlin, similar to those of Tate's Nonlin, are not described in detail here, and one should refer to Tate's technical report for more information.

Given a set of operator/action schemas specified by a "knowledge engineer" for a given planning domain, Nonlin can generate a plan for a domain problem consisting of an initial state and one or more (conjunctive) goals specified by a user. Domain operators, states, and goals are specified via Nonlin's *Task Formalism*, described in Section 2.

Nonlin is hierarchical in that actions to achieve goals can be specified as hierarchies with abstraction. These hierarchical abstractions will be used by Nonlin to develop a plan that is first composed of higher-level operators and is progressively refined into lower-level operators and ultimately into primitive (i.e., directly executable) actions.

Nonlin can plan for problems in which there may be multiple goals that interact where planning by generating a *linear* sequence of actions might be inadequate. To plan in a *nonlinear* manner, Nonlin represents plans as a *partially-ordered* network of actions. A least-commitment strategy is used which avoids introducing an ordering among actions unless it is necessary. Such an ordering may be necessary when one action is needed to establish a precondition for another action or when the effects of multiple actions *interact* (e.g., when one action denies a precondition for another action). When the planning is completed, a totally-ordered plan (i.e., one of possibly several total orderings) is output.

Nonlin is capable of *backtracking* to produce an alternate plan (*replanning*) and backtracking when it reaches a dead end during planning. Choice points (and their corresponding contexts, see Section 3.3) are preserved in a stack whenever Nonlin selects between two or more alternatives. Choices may be made during schema selection, schema instantiation, and action linearization. Backtracking to choice points/contexts is done in a depth-first manner.

The basic control cycle of the planner consists of expanding the current nodes in the network and checking for interactions between actions (and correcting any by linearization) until the network is fully expanded and the plan is generated. If at any point the plan cannot be expanded or an interaction cannot be corrected then the planner backtracks (if possible), switches contexts, and continues.

A Question-Answering (q&a) component of Nonlin is used to determine the value(s) of a proposition at a node in the network (in a given context). The answer returned by QA is not fully deterministic since the network is a partial ordering. The q&a component is also used to determine what links (linearizations) could be added to the network to establish a given value of a proposition at a node.

Input to Nonlin consists of:

- the Task Formalism definitions of the operators and actions for a domain (these definitions are actually incorporated into the source code)
- a list of facts that are always true
- a list of facts describing the initial state of the world
- a list of goals (states) to be achieved

Output from Nonlin consists of:

- a plan (alternate plans can be generated) [1]
- the optional output of the various program data structures

[1] The plan is printed to the user and also output via several global variables (see Section 3.4).

1.1 The Rest of This Document

The following sections of the manual describe how to define problem domains (Section 2), what the major components of Nonlin are (Section 3), and how to install and run Nonlin (Section 4). Some caveats about this implementation are given in Section 3.5. Information on source files, sample schema definitions and runs, and additional tips for defining schemas are given in the appendices.

1.2 Support for UM Nonlin

This project is being distributed free of charge with no implied warranty or support. If you FTP UM Nonlin, please let us know by sending mail to nonlin-users-request@cs.umd.edu (so we can put you on our mailing list, etc.).

The following internet email mailing lists have been set up to allow communication among users of UM Nonlin:

- **nonlin-users@cs.umd.edu** – a general mailing list of Nonlin users that is primarily intended for use by the UM Nonlin maintainers to announce new releases, fixes, etc.
- **nonlin-users-request@cs.umd.edu** – for requesting information from the UM Nonlin maintainers. Send mail here to be added to or deleted from the nonlin-users mailing list.
- **nonlin-bugs@cs.umd.edu** – for reporting bugs to the UM Nonlin maintainers. Bugs will be investigated as time permits.

1.2.1 Acknowledgements

The implementation of UM Nonlin was done by Subrata Ghosh and Rao Kambhampati while they were graduate students working with Prof. James Hendler. Subsequent maintenance of the code and the creation of this user manual was done by Brian Kettler. Prof. Austin Tate has provided useful information about the original Nonlin. Thanks also to the users that have provided bug reports and other feedback.

2 Defining Problem Domains: The Task Formalism

The Task Formalism (TF) is Tate's mechanism for defining the operators and actions of the problem domain. TF draws from Sacerdoti's SOUP (Semantics of User Program) language and uses the STRIPS representation of operators.

The problem domain can be decomposed into higher and lower level operators. The planner develops the plan by considering operators that match the higher level goals. These high level operators are later expanded by the planner into lower level operators. Eventually these are expanded into primitive (i.e., non-decomposable) actions. Decomposing the domain can make the planner more efficient. The decomposition of operators is specified by the TF definitions for the problem. Decomposition can take place to an arbitrary depth.

TF definitions are placed in the file **operators.lisp**. A subset of Tate's TF has been implemented here. The primary TF entities are **ACTSCHEMA** and **OPSCHEMA** which have identical syntax and semantics. By convention OPSCHEMAs are used to represent how a particular goal is achieved (i.e., they expand into subgoals and actions). ACTSCHEMAs are used to represent actions which have **effects**. Tate briefly mentions this distinction but does not motivate it in detail.

Schemas may have the following parts:

type: "ACTSCHEMA" or "OPSCHEMA"

name: schema name

todo: This is the pattern which typically includes variables. When expanding a goal node Nonlin attempts to unify this pattern with the goal. If there is a match then this schema is a candidate for instantiation.

expansion: The expansion of the schema. This is a partially-ordered list of steps. Each step will become a node in TASKQUEUE after the expansion. Every step has the following fields: id (usually "stepN"), type (one of "goal, action, primitive"), and pattern. The **type** field indicates the type of node resulting from the expansion of that step in the schema. The node types differ in how they are expanded by the planner (see Section 3.2). Note that the **id** field is used only within the schema definition. When the schema is expanded the system will assign its own id and node number to each resulting node.

orderings: The ordering(s) among steps in the expansion. Each ordering is of the form `m -> n` where m, n are the ids of two steps. Orderings are transitive. Orderings may be left unspecified between steps.

conditions: A condition determines whether an applicable schema (i.e., one whose pattern unifies with the current subgoal) can be instantiated. Each condition has the following fields: type (one of "precond, unsuperv, use-when, use-only-for-query" - see descriptions below), pattern (negated patterns are unsupported), purpose (**at** `step-id`), and contributor (**from** `step-id`). The **from** field is only for conditions of type **precond**. The step specified in the **from** field will be listed as a contributor for the step specified in the **to** field.[2] The types of conditions are mentioned below. Some precondition-type conditions may be added automatically by Nonlin as described below.

effects: The effects of the action. This includes **assert** and **delete** statements (predicates containing variables).

variables: This is a list of variables used in the schema definition.

The syntax for schema definition is very similar to that specified by Tate. Sample schemas are given in Appendix B.

Some terminology differences between the types of **conditions** in Tate's TF and TF as implemented here are:

1. **:precond** type conditions (preconditions) are equivalent to Tate's **supervised** conditions. The planner achieves these conditions in order to apply an operator. In the process the plan is expanded to establish these conditions.

2. **:unsuperv** type conditions (unsupervised) are implemented as described in Tate's report. Unsupervised conditions are preconditions to be established before an operator can be applied. Unlike regular preconditions (those of type **:precond**), however, unsupervised conditions are established by some other, unspecified schema and thus do not result in the expansion of the plan. The planner will not attempt to achieve these unsupervised conditions until the entire plan has been expanded. It will then attempt to achieve all of the unsupervised conditions (via linearizations - not expansions). If an unsupervised

[2] Note, Nonlin does not verify that the from-step actually asserts the condition for the to-step. Thus the user should insure that these fields are correct.

condition cannot be achieved, the planner will backtrack. If any of these conditions cannot ultimately be achieved, the planner will fail to generate a plan. For an example of unsupervised conditions see the sample house building schema in Appendix B.

3. **:use-when** type conditions are equivalent to Tate's **holds** conditions and are also known as *filter* conditions. They are used to determine the relevancy of an operator. The plan is not expanded as a result of these conditions. These conditions are used for schema selection and once satisfied are never removed as long as the schema instantiation is not removed. *See the note on :use-when conditions in the Caveats List (Section 3.5).*

   There are two special :use-when conditions: `(equal` var_1 var_2`)` and `(not (equal` var_1 var_2`))`. They are used to specify that two variables that appear in a schema must codesignate and must not codesignate, respectively. These conditions should be listed *after* any other conditions in the schema that may bind the specified variables. Note that **equal** is the *only* predicate that may be negated in a condition.[3]

4. **use-only-for-query** type conditions are not described in Tate's technical report because they were a later addition to Nonlin. In Nonlin all patterns in the plan state (e.g., action patterns, goal patterns, effects of instantiated schemas, etc.) must be ground (i.e., all of their variables bound).

   :use-only-for-query conditions allow their variables to be *temporarily* bound. If necessary, such a binding can later be changed (in contrast to bindings for other types of conditions which are protected). Such a binding may be changed, for example, when an interaction cannot be eliminated by a linearization (see the function **try-to-modify-binding** and other functions in the file **link.lisp**). *See the note on :use-only-for-query conditions in the Caveats List, Section 3.5).*

   For an example of where this type of condition would be used refer to the blocks world TF given in Appendix B. In the **puton** actschema

[3]Tate's Nonlin, in contrast, allows arbitrary predicates to be negated in conditions. To avoid issues about the meaning of negated predicates (and to simplify the implementation), UM Nonlin does not permit this. In most cases negated predicates can be specified using an additional (non-negated) predicate. For example, the predicate `closed` could be used as the negation of predicate `open`. Care must be taken that any action asserting `(closed` x`)` should also delete `(open` x`)`, and vice versa.

the condition **(on ?x ?z)** exists so that the variable z will be bound to the block *currently* beneath block x. Let's assume that x = Block A and z = Block B, and thus the condition is **(on A B)**. The variable is also bound to Block B in the effects of the schema. The binding of z to a *particular* block (block B in this case) is not a condition for the use of the schema. That is, z can be any block which x happens to be on top of at the time. During the course of further planning Block A may need to placed on top of another block. If the condition **(on A B)** were protected then the planner would have a problem (i.e., an interaction would occur). But since it is a **use-only-for-query** condition, the planner can rebind variable z to be whatever is necessary to continue planning.

The value of the global variable ***autocond*** determines if preconditions will be automatically added to the schema without the need to specify such conditions explicitly in the schema definition. If ***autocond*** = T or **AUTO** is specified in the schema definition then for each step in the expansion of type **goal** (followed by a step of type **action**) a precondition of the form **(:precond** `pattern` **:at** `pnode` **:from** `cnode`**)** will be added where "pattern" matches the goal step pattern, "pnode" matches the node number assigned to the node from expansion of the action step, and "cnode" is the node number from the expansion of the goal step. For example, in schema MAKEON given in Appendix B the preconditions **(:precond (cleartop ?x) :at step3 :from step1)** and **(:precond (cleartop ?y) :at step3 :from step1)** would automatically be added (if ***autocond*** = T). To disable this automatic behavior globally, set ***autocond*** to NIL. To disable it for a particular schema only, specify **NONAUTO** in the condition section of the schema.

Some minor TF statements for schemas have not been implemented here. These include: COMPUTE conditions, MAINEFFECTS, LEVELS, COST. These are a possible target for future implementation.

Schemas are stored in a **Schema Table** which is a hash table where schemas are hashed on their **pattern (todo)** field.

See Appendix B for some sample schema definitions.

3 Inside UM Nonlin

This section briefly describes some aspects of the UM Nonlin program's design. It is not intended to be comprehensive but rather to point out the major data structures, etc. that might be useful to know about when debugging a domain definition, tracing Nonlin's solution of a plan, etc.

3.1 Major Program Data Structures

The planner uses the following major data structures during planning. Most of these can be traced during Nonlin's operation (see section 4.5).

- **ALLNODES**

  This contains the partially ordered network of actions as a collection of nodes. ALLNODES is implemented as an array. Particular nodes are referenced by their unique node numbers. Each node is implemented as a Common Lisp structure with various fields: **id, nodenum, type, parent, children, prenodes, succnodes, expansion, expandconds, expanded, ctxt (effects), mark, nodevars**. These fields have the same meaning as specified by Tate.

  The file **def.lisp** contains routines for node definition and access.

- **GOST**

  The GOST (Goal Structure Table) is used for resolving interactions (through linearization) in the developing plan. Entries in the GOST give the purpose (if any) of a particular effect at any node. Conditions on the nodes are stored in the GOST together with lists of contributors. The GOST is implemented as a hash table, and the conditions are hashed on their pattern.

  The file **gost.lisp** contains GOST routines.

- **TOME**

  The TOME (Table of Multiple Effects) is used for detecting interactions in the developing plan. Effects at each node are stored in the TOME. The TOME is also used by the q&a component to determine if a proposition is true at a given point in the network. It is also implemented as a hash table, and the effects are hashed on their pattern.

  The file **tome.lisp** contains TOME routines.

- **TASKQUEUE**

  This contains all the nodes which are yet to be expanded. The planner terminates when the TASKQUEUE becomes empty. Currently it is implemented as a LIFO list of nodes.

The file **def.lisp** contains TASKQUEUE routines.

- **CONTEXT-LIST**

  This contains all backtracking choices. It lists backtracking points and the alternatives available at each point.

  The file **backtrack.lisp** contains CONTEXT-LIST routines.

3.2 Network Node Types

The following are the types of nodes in the network (ALLNODES):

- **GOAL** nodes are expandable. The planner first determines whether the goal (pattern) is already true in the current context. If it is true then it is replaced by a **PHANTOM** node. Otherwise the planner will attempt to establish the goal.

- **PHANTOM** nodes are not expandable. They are used as placeholders for facts that have already been established in the current context. However if a phantom node's fact (pattern) later becomes false then the phantom node reverts to being a goal node (which will then be placed on the task queue).

- **ACTION** nodes are expandable unless they represent **primitive** actions. The planner will not, however, attempt to see if the pattern of an ACTION node is already established in the current context (as it does for GOAL nodes).

- **PLANHEAD, DUMMY** nodes are not expandable. They are used to specify the network structure and serve as points where conditions may be attached to (for the planner's internal use only).

.

3.3 The Context Mechanism

This is used to implement the backtracking feature of Nonlin described previously. A context variable (generated by **gensym**) is added to every node, link, condition and effect. A new context is generated every time a backtracking point (caused by making a choice between multiple schemas or multiple linearizations) is reached. One of the alternatives is selected, and the rest are saved in the ***context-list*** with the corresponding context. Whenever the planner fails, the backtracking routine (in file **backtrack.lisp**) backs up to an earlier context where there is some alternative choice. It throws away everything (nodes, links, conditions and effects) that were generated after the context it backtracks to. The planner selects one of the alternatives and continues.

3.4 Some Useful Global Variables

The following can be set by the user to control Nonlin:

Variable	**Description**
autocond*	when T (NIL) enables (disables) automatic condition generation
(see previous section)	
cycle-limit*	the max. number of planning cycles (node expansion cycles) to run

The following global variables contain output from Nonlin after it has successfully terminated:

Variable	**Description**
cycle-count*	number of planning cycles (node expansions) done
planner-in*	input given to planner
planner-out*	output from planner (ordered list of primitive actions)
nonlin-termin-status*	status of Nonlin at termination

3.5 Some Caveats

There are some known "problems" with UM Nonlin. Most of these are not bugs per se but rather originate from the design of the current implementation. Some have work-arounds and others can be avoided. Most of these are subtle and should not arise frequently. At some point we hope to correct these, but as most require significant redesign and reimplementation, they will not be remedied in the near term.

Matching of facts in initial state are handled incorrectly, which can negatively impacts completeness of plan generation. A :use-when type condition is matched first against the always context and then to the initial context (state). If a match to the condition is found, ONLY variable bindings from the initial state facts matched will be returned. That is, the condition will NOT be matched to the effects of other potential contributors (i.e., the general Q&A match procedure will not be invoked.). Thus other contributors will be ignored and problems will ensue if a clobberer node is placed between the initial state and the condition's consumer. It is unclear why Nonlin was done this way (see cryptic comment in Q&A (in establish.lisp). This problem is only relevant for facts that can be contributed by both the initial context (or always context) AND by effects of actions in the plan. For facts of this sort, the user could have every plan expand into a dummy action which asserts the facts in question, rather than having them asserted in the initial or always contexts.

:use-only-for-query conditions may be improperly re-bound, negatively impacting plan correctness. A :use-only-for-query condition can be re-bound if necessary. If a condition C1 of this type contains variable X, where X appears in another condition C2 of the same schema instantiation that introduced C, it is possible for X to later be re-bound in C1. When this occurs, however, X will not get re-bound in C2 to its new value. Hence variable X will wrongly have the newer value in C1 but the older value in C2. Note that C2 can be a condition of any type (e.g., :use-when, etc.). The current "fix" is to avoid the use of :use-only-for-query conditions that contain a variable used in other conditions.

All conditions must be NECESSARILY (vs. possibly) true to be considered established, which can Negatively impacts plan completeness. When Q&A is called on condition C of node N, C must be true at N otherwise C will be considered unachieved/failing. E.g., if a potential contributor N+ exists for C and a potential clobberer N- exists for C, N+ and N- precede N, and N+ and N- are in parallel to each other, C will not be necessarily true and will fail. Nonlin will NOT try to order N- before N+ so as to achieve C for N. No fix exists for this currently. For now, user could put more ordering constraints into the schema definitions to reduce the number of tasks in parallel and/or reduce the use of goals in parallel that have the same predicate.

4 Running Nonlin

The following is the general procedure for running Nonlin. Details are given in subsequent sections.

1. Install Nonlin before using it for the first time.
2. Define the planning domain (if required).
3. Load Nonlin before each session.
4. Invoke Nonlin for each planning problem.

4.1 Installing Nonlin

To install this at your site:

1. FTP the *binary* file `nonlin-files.tar.Z` to your system via anonymous ftp from cs.umd.edu (directory /pub/nonlin).
2. Uncompress the compressed files (i.e., those with a ".Z" suffix) via `uncompress nonlin-files.tar.Z`.
3. The source code files should be extracted from the file **nonlin-files.tar** via
   `tar -xvf nonlin-files.tar`. This will install the files in a subdirectory named **Nonlin**
4. Edit the value set for the directory pathname and lisp binary file extension in **load-nonlin.lisp**.

The Nonlin code is loaded into a lisp package called :nonlin.

The user manual describing the implementation and how to run Nonlin is contained in the files **nonlin.ps** (postscript version), **nonlin.dvi** (device-independent version), and **nonlin.tex** (LaTex source). (These document files should be first uncompressed as described above).

The file **README** contains miscellaneous notes on the implementation.

The file **load-nonlin.lisp** contains Lisp functions for loading and compiling the planner files. The user should ensure that the file pathnames specified in this file point to the directory in which Nonlin has been installed.

4.2 Defining the Domain

(This step is not required if you want to use an existing domain that someone else has defined.)

Create a lisp source file containing the operators for your domain specified in Task Formalism as described in Section 2. See the file **blocks-operators.lisp** for an example (a listing of this file can be found in Appendix B).

4.3 Loading Nonlin

To load Nonlin before each session do the following. (Note that some of these commands may vary depending on the specific Common Lisp implementation being used.)

1. Invoke your common lisp.
2. If necessary load the file **load-nonlin.lisp** via `(load "load-nonlin.lisp")`.
3. The planner source files are loaded via `(load-nonlin-sources)`. Note that this step may be omitted if the sources have already been compiled (see below).
4. The planner source files are compiled via `(compile-nonlin-sources)`. Note that this step may be omitted if binary object files for the program already exist.
5. The planner object files are loaded via `(load-nonlin-binaries)`.
6. Load the lisp file containing the operators for your domain (e.g., blocks-operators.lisp for blocks world).
7. Optionally load a lisp file containing predefined planning problems for your domain (e.g., blocks-sample-probs.lisp for blocks world).

You may wish to invoke **(reset-schematable)** prior to to (re)loading a new set of schemas. You could also place a statement in the operators file itself to invoke that function.

4.4 Invoking Nonlin

After Nonlin is loaded it is invoked for each planning problem. Planning problems may be supplied interactively by the user or may already be defined. To plan for a *new* problem:

1. Start the planner via `(plan-for)`.
2. Enter the problem name when prompted (any valid Lisp atom is acceptable).
3. Enter a list of facts (first order formulae) that are always true. A fact or goal is a first-order formula of the form ($p\ arg_1\ arg_2 \ldots arg_n$), where p is a predicate (e.g., `(on a b)`).
4. Enter a list of facts of the initial state.
5. Enter a list of goals to be achieved.

To plan for a *predefined* problem (i.e., a problem that has been defined in a lisp file, e.g. blocks-sample-probs.lisp) invoke the **plan-for** function with the name of the problem, i.e. `(plan-for :problem '<problem-name>)`. For example, after loading the files blocks-operators.lisp and blocks-sample-probs.lisp, you can test Nonlin by having it solve the predefined problem sussman-blocks (Sussman's anomaly) by typing: `(plan-for :problem 'sussman-blocks)`.

If the planner finds a successful plan, it prints out the plan as a sequence of primitive operations and then prints the message `Replan ?` and waits for an input. If the user answers **yes** it continues to find alternative plans[4]. When no alternatives are left the message "`no more solutions`" is printed out and Nonlin terminates.

See Appendix D for some sample runs in the blocks world domain.

4.5 Debugging Options

The planner can be run in verbose mode with the **debug** option. A user can trace one or more of the major data structures (e.g., the GOST, TOME, etc. – see Section 3.1) using the functions **nonlin-debug** (enables debugging) and **nonlin-undebug** (disable debugging). Help on these functions is available by invoking the function **nonlin-debug-options**.

There are other functions that assist in debugging. Many of these are defined in the file **def.lisp**. These include functions for displaying data structures, initializing them, etc. Two of these are **reset-schematable**, which resets the schema table (after which you must reload the operators

[4]Note that **Replan** backtracks to the latest choice point, selects a new alternative and continues from there. But that does not always generate a new plan.

file you are using), and **dump-schematable**, which displays the contents of the schema table (i.e., the operators that have been loaded).

A A Brief Description of the Nonlin Files

backtrack.lisp contains backtracking routines.

def.lisp contains most of the data structure definitions, initializations and access routines

establish.lisp The main function in this file is **try-to-establish**. For **goal** nodes this function tries to establish the condition to be achieved by the goal node. With the help of **q&a-process-tome-entry**, the function **q&a** checks if a condition is true at a particular node. Q&a accepts non-ground patterns. **q&a-process-tome-entry** accepts only ground patterns and using the tome it computes the four critical lists (**VL, PARVL, VNOTL and PARVNOTL**) as defined in [TATE76] and returns them to the calling function (**q&a**). When called with a non-ground pattern **q&a** finds all possible bindings and calls **q&a-process-tome-entry** with each binding

expand.lisp contains the schema expansion routine. Given a schema and a node, this function replaces the node by its expansion. It does all relocation, additions of effects and conditions.

gost.lisp contains gost-entry definition and gost access (lookup, insertion and deletion) routines.

init-planner.lisp contains the function **get-problem** which gets the input problem interactively from the user and store/access predicates that are always true, stored in ***always-ctxt*** (node number -2) and the predicates that are initially true, stored in ***init-ctxt*** (node number -1)

load-nonlin.lisp contains functions for loading program files. It contains the directory path names for the program files.

link.lisp contains routines to detect and correct interactions. The function **nonlin-link** takes an interact list, makes all possible pair of interactions and then removes the interactions by successively calling **resolve-link**. Given a pair of interactions **resolve-link** computes all possible linearizations.

mark.lisp contains functions to mark the network (i.e mark each node **:before :after** or **:parallel**) with respect to a given node passed as a parameter

plan.lisp contains the top level control structure of Nonlin. The function **planner()** picks a node from **TASKQUEUE** each iteration and expands that node.

printplan.lisp contains a topological sort routine which prints out one of the many possible linear plans from the partially ordered network in **ALLNODES**.

readschema.lisp contains routines that process the schema definitions.

schema.lisp contains the function **select-schema-to-expand** which finds all applicable schemas from schematable, finds all possible bindings and with the schemas and the bindings makes all possible ground schema instances. It saves all but one schemas in the backtracking list and returns one schema to the calling routine.

tome.lisp contains tome-entry definition and tome access functions.

unify.lisp contains all unification routines.

util.lisp contains some utility macros.

blocks-operators.lisp contains schema definitions for the blocks world domain.

blocks-sample-probs.lisp contains predefined sample problems for the blocks world domain.

house-operators.lisp contains schema definitions for the Tate's house building domain.

B Sample Schema Definitions

B.1 Blocks World Domain

The following operators implement a small blocks world domain. These operator schemas are in the file **blocks-operators.lisp**. For additional tips on specifying schemas, see the next appendix.

```
; The following statement will cause preconditions to be added to the
; schema without having to explicitly specify them.
(setf *autocond* t)

; The following schema details how the goal of (on ?x ?y) is to be
; achieved.  There are 2 preconditions (pc1, pc2) to be achieved
; before doing an action to achieve (puton ?x ?y).
; Note that no ordering is specified between pc1 and pc2.
; Specifying orderings where possible improves planner efficiency,
; but orderings should NOT be specified so as to over-constrain
; the planner.
; Also, by specifying pc1 and pc2 as GOALS, rather than as
; ACTIONS, the planner will first check to see if they are
; already achieved BEFORE expanding them.  On the other hand,
; since we know that ?x is not on ?y (since this schema would
; not be selected to achieve an already achieved goal),
; we can tell the planner to find an action to achieve (puton ?x ?y)
; without bothering to check first to see if it is necessary.

(opschema makeon
    :todo (on ?x ?y)
    :expansion (
                (pc1 :goal (cleartop ?x))   ; pc1, pc2, act are
                (pc2 :goal (cleartop ?y))   ; arbitrary labels
                (act :action (puton ?x ?y))
               )
    :orderings ((pc1 -> act) (pc2 -> act))
    :variables (?x ?y)
)

; The following conditions will be added by the planner to the above schema
```

```
; (since *autocond* is enabled):
;   :conditions ((:precond (cleartop ?x) :at act :from pc1)
;                (:precond (cleartop ?y) :at act :from pc2))


; The following schema details how to achieve (cleartop ?x).
; Here pc1 is a precondition to be achieved before doing an action
; to achieve act.  This schema contains filter (:use-when) conditions
; which, if not met, will keep this schema from being instantiated.
; The first condition is used to match ?y with what is now on top of ?x.
; The second condition is used to find a block (or table) which is
; clear, and thus ?y can be removed from atop ?x and put on ?z
; This is achieved by first clearing ?y (pc1) and then putting ?y
; on ?z.  In order to do this ?x, ?y, and ?z must be distinct blocks
; (filter conditions 3 and 4).  Remember that filter conditions must
; be satisfied in the current context.  The planner will NOT attempt
; to satisfy them (as it would for preconditions such as pc1).

(opschema makeclear
    :todo (cleartop ?x)
    :expansion (
               (pc1 :goal (cleartop ?y))
               (act :action (puton ?y ?z))
              )
    :orderings ((pc1 -> act))
    :conditions (
                (:use-when (on ?y ?x) :at act)
                (:use-when (cleartop ?z) :at act)
                (:use-when (not (equal ?z ?y)) :at pc1)
                (:use-when (not (equal ?x ?z)) :at pc1)
               )
    :variables (?x ?y ?z)
)


; The following conditions will be added by the planner to the above schema
; (since *autocond* is enabled):
;   :conditions ((:precond (cleartop ?y) :at act :from pc1))
```

```
; The following schema achieves (puton ?x ?y) by the primitive action
; puton-action.  There are no preconditions and 3 filter conditions.
; The first 2 filter conditions say that before applying this schema,
; (cleartop ?x) and (cleartop ?y) must have been achieved.  The orderings
; in the makeon schema will ensure this, but stating these conditions
; explicitly here protects them from being clobbered (by an action that
; comes between the actions that establish these preconditions and
; and puton-action).  The third filter condition is used to bind
; ?z for purposes of specifying the effects of the action.  (See the
; section in the text on :use-only-for-query conditions for a further
; description of this example).
; Note also that this schema has effects (of the primitive action
; puton-action).  Thus this schema is an ``actschema'' instead of
; an ``opschema'', although this labeling makes no difference to Nonlin.

(actschema puton
    :todo (puton ?x ?y)
    :expansion  ((act :primitive (puton-action ?x ?y)))
                ; this action is primitive (and hence non-expandable)
    :conditions (
                 (:use-when (cleartop ?x) :at act)
                 (:use-when (cleartop ?y) :at act)
                 (:use-only-for-query (on ?x ?z) :at act)
                )
    :effects    ((act :assert (on ?x ?y))
                 (act :assert (cleartop ?z))
                 (act :delete (cleartop ?y))
                 (act :delete (on ?x ?z)))
    :variables (?x ?y ?z)
)
```

B.2 House Building Domain

This example is intended to illustrate how one of the schema definitions for house building, given in Tate's technical report, would look in the Task Formalism as it has been implemented here. This schema is from page 6 [TATE76]. Below is how it looks for this implementation. See the file **house-operators.lisp** for the rest of the house building schemas.

To plan in this domain:

1. Clear the schema table via `(reset-schematable)`
2. Load the operators via `(load "operators-tate.lisp")`
3. Invoke the planner via `(plan-for)`
4. Enter `house` for `Name of the Problem?`
5. Enter `()` for the next two prompts for `Facts...`
6. Enter `((build house))` for `Goals to be achieved...`

```
; The following schema illustrates how actions can be specified
; hierarchically (i.e., goal (decorate) decomposes into these
; 6 actions).  These actions are in turn decomposed into other
; actions.  The (partial) orderings between these actions that have
; been specified constrain the search space of the planner and
; thus improve planning efficiency.

(actschema decor
  :todo (decorate)
  :expansion (
             (step1 :action (fasten plaster and plaster board))
             (step2 :action (pour basement floor            ))
             (step3 :action (lay finished flooring          ))
             (step4 :action (finish carpentry               ))
             (step5 :action (sand and varnish floors        ))
             (step6 :action (paint                          ))
            )
  :orderings ( (step2 -> step3) (step3 -> step4) (step4 -> step5)
              (step1 -> step3)  (step6 -> step5))
  :conditions (
              (:unsuperv (rough plumbing installed   ) :at step1)
              (:unsuperv (rough wiring installed     ) :at step1)
              (:unsuperv (air conditioning installed ) :at step1)
              (:unsuperv (drains installed           ) :at step2)
              (:unsuperv (plumbing finished          ) :at step6)
              (:unsuperv (kitchen equipment installed) :at step6)
              (:precond  (plastering finished        ) :at step3 :from step1)
              (:precond  (basement floor laid        ) :at step3 :from step2)
              (:precond  (flooring finished          ) :at step4 :from step3)
              (:precond  (carpentry finished         ) :at step5 :from step4)
              (:precond  (painted                    ) :at step5 :from step6)
             )
)
```

C Tips for Working with Operator Schemas

The following tips are meant to assist in the (somewhat arcane) "art" of schema definition. There are of course many ways that operators can be specified for a given domain. Some will be more efficient than others. A poorly specified schema, however, can result in it being impossible for the planner to generate a plan. The best way to define good schemas is to experiment by tracing the planner's operation with the schemas (see the section on debugging).

Users of this program are encouraged to submit their own tips via email to nonlin-users-request@cs.umd.edu.

C.1 Bound Variables and Schema Instantiation

This tip was submitted by Subbarao Kambhampati (Arizona State University).

The most tricky thing that needs to be specifically stated in Nonlin manual is that Nonlin isn't guaranteed to work when all the variables are not bound at the schema instantiation time.

The two ways of binding variables are (1) by putting the variable in the :todo field, and (2) by putting the variable in a :use-when condition.

Suppose you have the goal (at robot ?y), and the way to achieve it is to move robot from ?x to ?y.

Novice users tend to write something like the following:

```
(opschema move-robot
   :todo (at robot ?y)

   :expansion (
               (step1 :goal (at robot ?x))
       (step2 :action (move robot ?x ?y))
              )

   :orderings ((step1 -> step2))
)
```

Now, this is almost certainly going to fail because Nonlin won't bind ?x at the time of schema selection, and thus it will be stuck with partially instantiated goals. Since the correspondences between the variables (the codesignation and noncodesignation constraints) are not maintained in a systematic fashion, once there are variables in the plan, all bets are off regarding completeness of the planner.

The right way to write this is to put a bunch of use-when conditions that will make sure that ?x will get bound before the schema is selected. That is:

```
(opschema move-robot
   :todo (at robot ?y)

   :expansion  ((step1 :action (move robot ?x ?y)))

   :conditions ((:use-when  (at robot ?x) :at step1))
)
```

D Sample Runs: Blocks World

```
IBUKI Common Lisp  release 01/01  October 15, 1987

PLANNER>(load-l)
NIL
PLANNER>(compile-l)
NIL
PLANNER>(load-o)
NIL
PLANNER>(plan-for)
Name of the Problem? '3blocks-problem
Facts always true (as a list)((cleartop table))
Facts of input situation (as a list)? ((on c a ) (on a b)(cleartop c)(on b table))
Goals to be achieved (as a list)? ((on a b)(on b c))
 The world state is ((ON B TABLE) (CLEARTOP C) (ON A B) (ON C A))

 the problem to be solved is
{SCH1016}
   '3BLOCKS-PROBLEM::PLAN
      Expansion:
           0 {<ND1008>[:DUMMY]}
           1 {<ND1009>[:GOAL(ON A B)]}
           2 {<ND1010>[:GOAL(ON B C)]}
           3 {<ND1011>[:DUMMY]}
      Conditions:
           <<SC1012>> :PRECOND (ON B C) :at 3  :from (2)
           <<SC1013>> :PRECOND (ON A B) :at 3  :from (1)
      Effects:
           <<SE1014>> :ASSERT (ON B C) :at 2
           <<SE1015>> :ASSERT (ON A B) :at 1
 ****The Planning is OVER
 The plan is...

***********INITIAL STATE************
(ON C A)
(ON A B)
(ON B TABLE)
(CLEARTOP C)
```

```
(CLEARTOP TABLE)
***************************************
 8: :PRIMITIVE (PUTON-ACTION C TABLE)   [Prenodes:(9)]          [Succnodes: (6)]
 6: :PRIMITIVE (PUTON-ACTION A TABLE)   [Prenodes:(8)]          [Succnodes: (4
                                                                             10)]
 4: :PRIMITIVE (PUTON-ACTION B C)       [Prenodes:(7 6)]        [Succnodes: (1
                                                                             3)]
 3: :PRIMITIVE (PUTON-ACTION A B)       [Prenodes:(12 11 4)]    [Succnodes: (1)]
***********GOAL STATE***************Replan  ?no
NIL

PLANNER>(debug '(allnodes))
(ALLNODES)

PLANNER>(plan-for)
Name of the Problem? '4blocks-2stack-problem
Facts always true (as a list)((cleartop table))
Facts of input situation (as a list)? ((on a b)(on b table)(cleartop a)
                                       (on c table)(on d table)(cleartop c)
                                       (cleartop d))
Goals to be achieved (as a list)? ((on a b)(on c d))
 The world state is ((CLEARTOP D) (CLEARTOP C) (ON D TABLE)
                     (ON C TABLE) (CLEARTOP A) (ON B TABLE) (ON A B))

 the problem to be solved is
{SCH67}
   '4BLOCKS-2STACK-PROBLEM::PLAN
      Expansion:
           0 {<ND59>[:DUMMY]}
           1 {<ND60>[:GOAL(ON A B)]}
           2 {<ND61>[:GOAL(ON C D)]}
           3 {<ND62>[:DUMMY]}
      Conditions:
           <<SC63>> :PRECOND (ON C D) :at 3  :from (2)
           <<SC64>> :PRECOND (ON A B) :at 3  :from (1)
      Effects:
           <<SE65>> :ASSERT (ON C D) :at 2
           <<SE66>> :ASSERT (ON A B) :at 1
expanding 3 :GOAL (ON A B) (Purpose (1))
```

```
after the expansion ..
CONTENTS of ALLNODES:
     0 {<ND68>[:PLANHEAD]{SUCC:(2)}}
     1 {<ND62>[:DUMMY]{PRE:(4 3)}}
     2 {<ND59>[:DUMMY]{PRE:(0)}{SUCC:(4 3)}}
     3 {<ND60>[:PHANTOM(ON A B)]{PRE:(2)}{SUCC:(1)}}
     4 {<ND61>[:GOAL(ON C D)]{PRE:(2)}{SUCC:(1)}}
expanding 4 :GOAL (ON C D) (Purpose (1))
after the expansion ..
CONTENTS of ALLNODES:
     0 {<ND68>[:PLANHEAD]{SUCC:(2)}}
     1 {<ND62>[:DUMMY]{PRE:(4 3)}}
     2 {<ND59>[:DUMMY]{PRE:(0)}{SUCC:(3 5)}}
     3 {<ND60>[:PHANTOM(ON A B)]{PRE:(2)}{SUCC:(1)}}
     4 {<ND921>[:ACTION(PUTON C D)]{PRE:(7 6)}{SUCC:(1)}}
     5 {<ND918>[:DUMMY]{PRE:(2)}{SUCC:(7 6)}}
     6 {<ND919>[:GOAL(CLEARTOP C)]{PRE:(5)}{SUCC:(4)}}
     7 {<ND920>[:GOAL(CLEARTOP D)]{PRE:(5)}{SUCC:(4)}}
expanding 6 :GOAL (CLEARTOP C) (Purpose (4))
after the expansion ..
CONTENTS of ALLNODES:
     0 {<ND68>[:PLANHEAD]{SUCC:(2)}}
     1 {<ND62>[:DUMMY]{PRE:(4 3)}}
     2 {<ND59>[:DUMMY]{PRE:(0)}{SUCC:(3 5)}}
     3 {<ND60>[:PHANTOM(ON A B)]{PRE:(2)}{SUCC:(1)}}
     4 {<ND921>[:ACTION(PUTON C D)]{PRE:(7 6)}{SUCC:(1)}}
     5 {<ND918>[:DUMMY]{PRE:(2)}{SUCC:(7 6)}}
     6 {<ND919>[:PHANTOM(CLEARTOP C)]{PRE:(5)}{SUCC:(4)}}
     7 {<ND920>[:GOAL(CLEARTOP D)]{PRE:(5)}{SUCC:(4)}}
expanding 7 :GOAL (CLEARTOP D) (Purpose (4))
after the expansion ..
CONTENTS of ALLNODES:
     0 {<ND68>[:PLANHEAD]{SUCC:(2)}}
     1 {<ND62>[:DUMMY]{PRE:(4 3)}}
     2 {<ND59>[:DUMMY]{PRE:(0)}{SUCC:(3 5)}}
     3 {<ND60>[:PHANTOM(ON A B)]{PRE:(2)}{SUCC:(1)}}
     4 {<ND921>[:ACTION(PUTON C D)]{PRE:(7 6)}{SUCC:(1)}}
     5 {<ND918>[:DUMMY]{PRE:(2)}{SUCC:(7 6)}}
     6 {<ND919>[:PHANTOM(CLEARTOP C)]{PRE:(5)}{SUCC:(4)}}
```

```
     7 {<ND920>[:PHANTOM(CLEARTOP D)]{PRE:(5)}{SUCC:(4)}}
expanding 4 :ACTION (PUTON C D) (Purpose NIL)
after the expansion ..
CONTENTS of ALLNODES:
     0 {<ND68>[:PLANHEAD]{SUCC:(2)}}
     1 {<ND62>[:DUMMY]{PRE:(4 3)}}
     2 {<ND59>[:DUMMY]{PRE:(0)}{SUCC:(3 5)}}
     3 {<ND60>[:PHANTOM(ON A B)]{PRE:(2)}{SUCC:(1)}}
     4 {<ND954>[:PRIMITIVE(PUTON-ACTION C D)]{PRE:(7 6)}{SUCC:(1)}}
     5 {<ND918>[:DUMMY]{PRE:(2)}{SUCC:(7 6)}}
     6 {<ND919>[:PHANTOM(CLEARTOP C)]{PRE:(5)}{SUCC:(4)}}
     7 {<ND920>[:PHANTOM(CLEARTOP D)]{PRE:(5)}{SUCC:(4)}}
 ****The Planning is OVER
 The plan is...

***********INITIAL STATE************
(CLEARTOP C)
(ON D TABLE)
(CLEARTOP A)
(ON B TABLE)
(ON A B)
(ON C TABLE)
(CLEARTOP D)
(CLEARTOP TABLE)

***************************************

 4: :PRIMITIVE (PUTON-ACTION C D)       [Prenodes:(7 6)]         [Succnodes: (1)]


***********GOAL STATE***************Replan  ?no
 The data structures are
CONTENTS of ALLNODES:
     0 {<ND68>[:PLANHEAD]{SUCC:(2)}}
     1 {<ND62>[:PLANTAIL]{PRE:(4 3)}}
     2 {<ND59>[:DUMMY]{PRE:(0)}{SUCC:(3 5)}}
     3 {<ND60>[:PHANTOM(ON A B)]{PRE:(2)}{SUCC:(1)}}
     4 {<ND954>[:PRIMITIVE(PUTON-ACTION C D)]{PRE:(7 6)}{SUCC:(1)}}
     5 {<ND918>[:DUMMY]{PRE:(2)}{SUCC:(7 6)}}
```

```
     6 {<ND919>[:PHANTOM(CLEARTOP C)]{PRE:(5)}{SUCC:(4)}}
     7 {<ND920>[:PHANTOM(CLEARTOP D)]{PRE:(5)}{SUCC:(4)}}
Gost Table Entrees:
Cond:(CLEARTOP C)  +[(<< :PHANTOM :at 6 :from (-1)>>
                      << :PRECOND :at 4 :from (6)>>) ]  -[NIL ]
Cond:(CLEARTOP D)  +[(<< :PHANTOM :at 7 :from (0)>>
                      << :PRECOND :at 4 :from (7)>>) ]  -[NIL ]
Cond:(ON C TABLE)  +[(<< :USE-ONLY-FOR-QUERY :at 4 :from (0)>>) ]  -[NIL ]
Cond:(ON A B)  +[(<< :PHANTOM :at 3 :from (-1)>>
                  << :PRECOND :at 1 :from (3)>>) ]  -[NIL ]
Cond:(ON C D)  +[(<< :PRECOND :at 1 :from (4)>>) ]  -[NIL ]

Tome Table Entrees:
Eff:(CLEARTOP C)  :assert[(6) ]  :delete[NIL ]
Eff:(CLEARTOP D)  :assert[(0 7) ]  :delete[(4) ]
Eff:(ON C TABLE)  :assert[(0) ]  :delete[(4) ]
Eff:(ON A B)  :assert[(3) ]  :delete[NIL ]
Eff:(ON C D)  :assert[(4) ]  :delete[NIL ]
NIL

PLANNER>(bye)
```

E Version History

E.1 Changes in Version 1.2.2 (11/92)

- put Nonlin code back into package :nonlin
- minor bug fixes
- replaced old loader INIT.LISP with LOAD-NONLIN.LISP
- renamed INIT-ALWAYS.LISP to BLOCKS-SAMPLE-PROBS.LISP
- renames OPERATORS.LISP to BLOCKS-OPERATORS.LISP
- replaced debug fn DEBUG with NONLIN-DEBUG, NONLIN-UNDEBUG, NONLIN-DEBUG-OPTIONS
- tested under Allegro CL 4.2 and MCL, V2.0
- added some documentation to code
- changes in User Manual

E.2 Changes in Version 1.2.0 (2/92)

The main purpose of this release is to fix several bugs reported by alert users. An effort was made to fix as many of these as possible. As a result, almost all of the Nonlin files contain changes. These changes have been tested on our Macintosh Common Lisp implementation and should work on other Common Lisps.

In addition, the PACKAGE statements have been removed from the source files as they have hindered the porting of the source code between different Lisps.

Nonlin now writes various info to some global variables. This was done to facilitate the use of Nonlin by other applications. These variables are:

- *cycle-count* records the number of node-expansion cycles
- *cycle-limit* is an optional limit on the number of these cycles
- *cycle-limit-p* is t if such a limit exists
- *nonlin-use-mode* is interactive (normal) if equal to t else is batch (i.e., no prompting to the user) if equal to nil

- *goals* records the goals input to the planner
- *planner-in* records the context and goals input to the planner
- *planner-out* records the output (plan) from the planner
- *kids* records a trace of node expansion