Contents

1 Overview ........................................ 5
   1.1 Introduction .................................. 5
   1.2 Longbow’s Distinguishing Features .......... 5

2 Installation ...................................... 6
   2.1 Obtaining Source Code ......................... 6
       2.1.1 Unix File Systems ...................... 7
       2.1.2 Non-Unix File Systems .................. 7
   2.2 Customizing the Initialization Files .......... 8
       2.2.1 General Modifications .................. 8
       2.2.2 The CLIM User Interface ............... 8
       2.2.3 Macintosh MCL 2.0 Graphics Interface .. 9
   2.3 Loading the Longbow System .................. 9
       2.3.1 Patch Files .................................. 10
   2.4 Loading a Longbow Domain .................... 10
   2.5 Running Longbow .................................. 10

3 Defining Longbow Planning Problems .......... 11
   3.1 The Condition Language ....................... 11
   3.2 Writing Action Operators ...................... 13
       3.2.1 The Parameter List ...................... 13
       3.2.2 The :primitive Slot ...................... 13
       3.2.3 Preconditions .......................... 13
       3.2.4 Constraints ............................. 15
       3.2.5 Effects .................................. 15
       3.2.6 Abstract Action Definitions ............. 16
   3.3 Writing Decomposition Operators ............... 16
       3.3.1 Parameters .................................. 19
       3.3.2 Constraints .................................. 19
       3.3.3 Steps ...................................... 21
       3.3.4 Rewrite Rules .............................. 22
       3.3.5 Causal Links .................................. 24
       3.3.6 Orderings .................................. 26
       3.3.7 The Partiality of Decomposition Operators .. 27
   3.4 Defining a Set-Up Function for a Planning Problem .. 28
A Predefined Planning Problems 53
B Longbow Operator Syntax 53
   B.1 Operator Syntax in EBNF Form ................................. 53
   B.2 Additional Constraints on Operators ............................ 56
      B.2.1 Action Operators ........................................ 56
      B.2.2 Decomposition Operators ................................ 56
C Functions for Interacting with Longbow 58
D Switches and Other Global Variables 67
   D.1 Switches ........................................................ 67
      D.1.1 Connection to the constraint-kb-query Function ........ 67
      D.1.2 Syntax Checking ........................................... 67
   D.2 Global Variables ................................................ 67
      D.2.1 Template-Storing Variables ............................... 67
      D.2.2 Plan-Space Bounds ....................................... 68
      D.2.3 Search State Variables .................................. 68
      D.2.4 Graphical User Interface Variables ..................... 68
E A Brief Guide to Longbow File Structure 69
   E.1 lb: Directory .................................................... 69
   E.2 lb:docs Directory ............................................... 70
   E.3 lb:mac-ui Directory ............................................. 70
   E.4 lb:clim-ui Directory ........................................... 71
   E.5 lb:contrib Directory .......................................... 71
   E.6 lb:code Directory .............................................. 71
   E.7 lb:ops Directory ............................................... 72
F Assorted Longbow Data Structures 73
   F.1 The plan Data Structure ....................................... 73
   F.2 The p-step Data Structure .................................... 74
   F.3 The effect Data Structure ..................................... 76
   F.4 The dlink Data Structure ..................................... 76
   F.5 The link Data Structure ...................................... 77
   F.6 The unsafe Data Structure ................................... 77
   F.7 The openc Data Structure .................................... 78
G Release Notes

H Proposed Future Enhancements
H.1 Unification ................................................................. 93
H.2 Operator Syntax ............................................................. 94
A Developer’s Guide to the
Longbow Discourse Planning System

Version 1.0

alpha 12

R. Michael Young

ISP Technical Report 94-5
Original Publication Date: December 1994
Current Revision Publication Date: September 10, 1996

Abstract

This document describes version 1.0 of the Longbow discourse planning system, including a description of the basic methods for installing Longbow on your system and loading and running planning problems. The Developer’s Guide also discusses the manner in which you can access some predefined example domains and describes how you can customize Longbow for your particular domain by defining your own operators planning problems and monitoring Longbow’s planning process as it attempts to solve these problems.
A Developer’s Guide to the Longbow Discourse Planning System

Version 1.0 alpha 12

R. Michael Young
Intelligent Systems Program
University of Pittsburgh
Pittsburgh, PA 15260
myoung+@pitt.edu

September 10, 1996
1 Overview

1.1 Introduction

Longbow is a hierarchical discourse planning system that combines decomposition reasoning (planning to perform an abstract action by performing that action’s substeps) and causal reasoning (planning to perform an action by ensuring that the action’s preconditions all hold before it’s execution) in the same representation.

The Longbow system is based on a domain-independent hierarchical AI planning algorithm named DPOCL (Decompositional, Partial-Order Causal Link planner) [10] which itself is built on top of a non-hierarchical planner, UCPOP [4]. Longbow differs from DPOCL and UCPOP by containing extensions that make representing communicative actions possible.

This document assumes some familiarity with discourse planning and partial order, causal link (POCL) planning described in the AI planning literature. For an overview of discourse planning, see [3]. An excellent overview of POCL-style planning is found in [6].

1.2 Longbow’s Distinguishing Features

Longbow is a hierarchical discourse planner the incorporates recent advances in AI Planning to provide a representation for discourse plans that is more expressive, has cleaner semantics, demonstrates more attractive formal properties and represents internal structure of discourse plans not described by previous systems.

Highlights of Longbow’s Feature List include:

- Longbow’s extended plan representation includes:
  - the intentional and informational structure of a discourse, represented in a clean and precise manner.
  - communicative actions with multiple effects.
  - tree-structured plans as well as graph-structured plans (that is, discourse plans that share components between distinct segments).
  - plans with interleaved subplans (that is, discourse plans that interleave communicative actions from distinct segments).
  - a flexible recipe-writing language allowing the operator writer to specify complete recipes, empty recipes or partial recipes.
Longbow’s planning algorithm provides:

- reasoning that combines hierarchical and causal planning.
- a plan-space search function, making debugging more straightforward.
- a provably sound and complete planner.
- the ability to include user-defined and domain-specific search-control knowledge distinct from the planning algorithm.
- easy integration with external knowledge bases.
- graphical user interfaces being developed for CLIM 2.0 and the Macintosh Common Lisp and ACL/PC user interface packages.

2 Installation

Longbow source code is written in accordance with the Common Lisp definition provided in Common Lisp: The Language, second edition, by Guy L. Steele [5]. It has been compiled and tested in Allegro Common Lisp and Lucid Common Lisp running on DECStation, Sun-4 and Hewlett-Packard workstations and in Macintosh Common Lisp 2.0 on Macintosh computers. The code is currently developed and maintained in Allegro Common Lisp version 4.2; support is provided for the implementation in Allegro Common Lisp and minimal support is available for other environments. Adherence to the standards defined in [5], however, should ensure that the code will run with little modification under other Common Lisp implementations and system architectures.

2.1 Obtaining Source Code

Longbow is currently being developed at the Intelligent Systems Program at the University of Pittsburgh. The source code for alpha versions of Longbow is available for limited release. If you are interested in becoming an alpha test site or are interested in receiving notice of new Longbow releases, please send email to longbow@isp.pitt.edu or contact R. Michael Young at

R. Michael Young
505 Learning Research and Development Center
University of Pittsburgh
Additional information is currently available in the Longbow Home Page, accessible on the World-Wide Web at the URL

http://www.isp.pitt.edu/~young/longbow

Current (and future) releases are (and will be) accessible from this page and via anonymous FTP in the file lb.tar.Z at www.isp.pitt.edu in the directory

`ftp/pub/young/longbow/current`.

All patches for the current release of Longbow will be available by anonymous ftp from the same machine in the directory

`ftp/pub/young/longbow/current/patches`.

A current version of this manual can be found in the same directory, in the file

`lb-manual.ps.Z`, along with an optional cover page in the file `lb-cover.ps.Z`. These files are in compressed postscript format.

### 2.1.1 Unix File Systems

The source code for Longbow is available for Unix machines in tar archive format. After obtaining the file `lb.tar.Z`, uncompress the tar file using the command

```
% uncompress lb.tar.Z
```

This command will create the file `lb.tar` in your current directory. To extract the Longbow source files from the tar archive, use the command

```
% tar -xvf lb.tar
```

This will create the subdirectory named `lb` in your current directory. This new directory is the top-level directory for all Longbow code.

### 2.1.2 Non-Unix File Systems

The file structure and individual files for non-Unix file systems are identical to those used in the Unix configuration. To obtain the Longbow files for non-Unix file systems
such as the Macintosh, Windows or DOS-based machines, extract the Longbow files on a Unix system as described above and copy the files, preserving the file structure, to the non-Unix system.

File systems with filename length limitations will need to have some Longbow file names shortened. Rename individual files as needed, making sure that all references to the file names in any Longbow source are edited to reflect this change. In particular, make certain that the contents of the file `lb:code;loader.lisp` is modified accordingly.

### 2.2 Customizing the Initialization Files

Modification of the initialization files is necessary to specify the correct path names that Longbow uses during file loading and compiling.

#### 2.2.1 General Modifications

Before loading Longbow code, create a logical pathname translation so that logical pathnames starting with `lb:` point to the top-level directory `lb` created when extracting the Longbow tar file. An example of the code that creates these translations is found in the file `lb-init.lisp`. You might want to add the contents of this file to your lisp init file while working with Longbow.

If your implementation does not support logical pathnames, substitute the complete pathname of the top-level Longbow directory for the logical pathname `lb:` when encountered throughout this document.

#### 2.2.2 The CLIM User Interface

The source code defining the CLIM 2.0 graphical user interface is found in the directory `lb:clim-ui`. To include these files in the Longbow system definition (and cause them to be accessed by `compile-longbow` and `load-longbow`), add the following form to your lisp init file:

```
(push :clim-2.0 *features*)
```

These files should only be included in the Longbow system definition if you are running Longbow in a Lisp environment in which the appropriate version of CLIM has
also been loaded. An error may result if these files are loaded, compiled or executed in any other environment.

2.2.3 Macintosh MCL 2.0 Graphics Interface

The source code defining the MCL 2.0 graphical user interface is found in the directory `1b:mac-ui`. To include these files in the Longbow system definition (and cause them to be accessed by `compile-longbow` and `load-longbow`), add the following form to your lisp init file:

```
(push :mac-ui *features*)
```

These files should only be included in the Longbow system definition if you are running Longbow on a Macintosh under Macintosh Common Lisp 2.0. An error may result if these files are loaded or compiled in any other environment.

2.3 Loading the Longbow System

To load the Longbow source code, first load the Longbow file `loader.lisp`:

```
> (load `'1b:code;loader.lisp')
```

This file contains a list of Longbow source code files and information about the order in which they must loaded.

Next, to compile the Longbow code for execution on your system, execute the following function:

```
> (compile-longbow)
```

This function will compile the Longbow code files and then load them into your lisp image. After compiling the Longbow files, subsequent loads can be performed without re-compiling by executing the function

```
> (load-longbow)
```

---

1 This file is automatically loaded if you load the file `1b-init.lisp` described above.
2.3.1 Patch Files

The current version of Longbow may also include a number of patchfiles, each adding some number of bug fixes or feature extensions. These files are available via anonymous ftp as described in section 2.1. All patch files should be loaded after all regular Longbow code is loaded. Patch files for a given release will be named patch1.lisp through patchn.lisp, indicating a numerical sequence from 1 to n. These files should be loaded in the sequence defined by the numbers that appear in their names.

Be sure only to load patchfiles associated with the current release. Loading patchfiles from previous releases may overwrite any subsequent updates or enhancements.

2.4 Loading a Longbow Domain

Defining the operators and associated functions for a particular domain is discussed in section 3. To load a domain once it has been defined, first load the file in which the associated domain operators and functions have been defined. 2 Longbow operator definitions are kept in the lb:ops/ directory. To load a file with name filename, execute the following call:

> (load 'lb:ops;filename')

For instance, a number of pre-defined sample domains are supplied with the release of Longbow. These domains are discussed in appendix A. To load these domains, execute the following function call:

> (load 'lb:ops;suite.lisp')

2.5 Running Longbow

Once a set of operators has been loaded, you’re ready to interact with the planner. First, change packages to the :1bow package used by the Longbow code:

> (in-package :1bow)

2Note that loading and running Longbow domains can also be performed from within the various graphical user interfaces described in section 6.
Then execute the problem’s set-up function:

> (setup-fn-name)

As discussed in section 3.4, a problem’s set-up function instantiates the planner’s global variables that contain the domain’s operator templates and initial and goal state descriptions.

Once the set-up function has been called, the planner can be run on a planning problem by executing the find-plan function, passing it the name of the planning problem you’re interested in.

> (find-plan problem-name)

The function find-plan is described further in appendix C. It returns two values: the plan located as a solution to the planning problem and a data structure storing statistics about the search process Longbow used to construct the plan.

Of course, you are free to define your own lisp function that calls the set-up function and find-plan together. An example of this type of user-defined function can be found in the file `lb:ops:suite.lisp` in the function test-describe3 as well as a number of other functions defined in this file.

3 Defining Longbow Planning Problems

Each time Longbow is run, the program solves a planning problem. A planning problem is a pairing of a set of operators with the specification of an initial state and a goal state. You will want to define your own planning problems for your domain. It’s easiest to do this by following the structure in the examples we’ve provided. In these examples, a planning problem is defined using a set-up function. The set-up function is responsible for defining the operators and the initial and final states. These components are described below, as is the approach to defining the set-up function.

3.1 The Condition Language

Longbow plans manipulate a representation of a domain. For most discourse problems, this domain will involve the mental state of the discourse participants. As the
system designer, you must specify the initial state of this domain, the goals of the planning problem and how operators change the domain to achieve the goal. Longbow provides a condition language, a language describing conditions in this domain, that you will use to build up your operators and planning problem specifications.

This condition language is described precisely in appendix B, but it’s worth looking at briefly here. The condition language is like the language used in a logical calculus. It has predicate and relation symbols with variable symbols that range over objects in the domain. Its variables also range over sentences in the logic as well.

Sentences consist of a list whose first element is a relation constant and whose remaining elements are object constants, variables or other sentences. Variables begin with the question mark symbol (?) and their bindings range over the finite collections of objects in the domain of discourse as well as over all sentences in the language.

The following are all valid sentences in the condition language:

(Pushed MAX ?someone)

(Cooperative ?Agent1 Robot14 ?task)

(Believes CLAUDIUS (Knows ?speaker ?proposition))

The following are examples of invalid sentences:

(?relation MIKE RENEE)

(Believe MIKE (?arrival-time) ?flight-1)

These sentences are invalid because a variable appears in the relation position (the first element of the list) in each of them.

One further restriction on the condition language is that no relation constant with the name forall may be used. This symbol is reserved for use in conditional component definitions (see section /refsec:constraints).
3.2 Writing Action Operators

An action operator describes a single action: the context required for its execution and the state of the world after the action has successfully completed. In Longbow, there are two types of actions. There are primitive actions that represent the directly executable actions in any domain. In hierarchical planners, primitive actions are those that form the leaf nodes of complete plans. Surface speech acts, for example, are typically represented as primitive actions in discourse planning systems.

Then there are abstract actions that represent entire subplans made up of more primitive actions. Abstract actions are the interior nodes in the plan graphs produced by hierarchical planning systems. In discourse planners, abstract actions have typically been used to represent the rhetorical structure or discourse relations that hold between actions or intentions in their subplans.

The syntax for Longbow’s action operators is given in appendix B. Each action operator has a number of slots, including a parameter list and lists of preconditions, constraints and effects. We describe the use of each of these slots below. Example action operators are shown in Figure 1.

3.2.1 The Parameter List

A parameter list is a list containing any number of regular variables and is used to enumerate all the variables that will be appear in the action operator.

3.2.2 The :primitive Slot

As described above, actions are either primitive or abstract. When an action operator’s primitive slot is set to t, the step is treated as primitive. When the slot is set to nil, the action is treated as abstract.

3.2.3 Preconditions

The preconditions of an action describe all the conditions that must hold so that the action will succeed when executed. When a precondition does not hold in a plan, Longbow will introduce new steps or use the effects of the initial state or steps already in the plan to make certain that the condition holds before execution.

\(^3\)Of course, there are potentially an infinite number of conditions that must be true before a step can successfully execute. System designers typically decide on a fixed number of conditions to model in their domains and simply ignore the rest.
(define (operator describe)
  :parameters (?x)
  :primitive nil
  :effect (know-about-hearer ?x))

(define (operator describe-class)
  :parameters (?x ?class)
  :primitive t
  :effect (know-about-hearer (class ?x ?class)))

(define (operator describe-has-part)
  :parameters (?object ?part)
  :primitive t
  :effect (know-about-hearer (has-part ?object ?part)))

(define (operator describe-has-step)
  :parameters (?process ?step)
  :primitive t
  :effect (know-about-hearer (has-step ?process ?step)))

Figure 1: Action Operators for Describing Objects and Processes

Longbow requires that the preconditions of an action must be internally consistent. Preconditions are listed in an action operator as a list, each element of the list being a single conjunct in an implicit conjunction.

When a step is added to a plan, all of its preconditions are open. Each open condition is added to the plan’s :flaws slot. Rather than require that all of a step’s preconditions be satisfied at one time, Longbow allows the function get-flaw to choose any flaw from a plan’s :flaws slot on each iteration of the planner. For more information about customizing get-flaw, see section 4.2.

To determine if a precondition can be satisfied from the initial state, one of two mechanisms can be used. First (and simplest), the developer can enumerate all the
facts that are true in the initial state by listing them in the problem definition (see section 3.4). Second, the developer can use a special user-defined function to satisfy the open condition. This function may be tied to a data base, a knowledge base or some other application-specific source for determining what conditions hold in the initial state. For more information about this user-defined function, see section 5

3.2.4 Constraints

The constraints of an action also describe conditions that must hold so that the action will succeed when executed. Unlike preconditions, however, Longbow will not spend any planning effort to ensure that a constraint holds before execution of a step. Constraints must be conditions that are satisfied only in the initial state. The relations that appear in constraints cannot appear in the preconditions or effects of any operators. At the time a step is first added to the plan, Longbow checks to see if all the step’s constraints are satisfied in the initial state. If they are not, then the step is not added. If they are, the step is added – after this, no further consideration is given to the step’s constraints.

Like preconditions, the constraints of an action must be internally consistent. Also like preconditions, constraints are listed in an action operator as a list, each element of the list being a single conjunct in an implicit conjunction.

To determine if a constraint can be satisfied from the initial state, one of two mechanisms can be used. First (and simplest), the developer can enumerate all the facts that are true in the initial state by listing them in the problem definition (see section 3.4). Second, the developer can use a special user-defined function to satisfy the open condition. This function may be linked to a data base, a knowledge base or some other application-specific source for determining what conditions hold in the initial state. For more information about this user-defined function, see section 5

3.2.5 Effects

The effects of an action describe all the conditions that will hold as a result of the action’s successful execution. Longbow requires that a step’s effects also be internally consistent and they are listed in an action operator as a list, each element of the list being a single conjunct in an implicit conjunction.
3.2.6 Abstract Action Definitions

Abstract actions represent abstractions of all the subplans that achieve some common goals from some common starting state. A good rule for writing the preconditions and effects of abstract actions is to include only those conditions that are needed as preconditions or achieved as goals of every subplan defined in the decomposition operators for that particular abstract action. The planner does not enforce this rule, but if operator writers ignore it by placing additional preconditions or effects in an abstract action, then the planner may spend unneeded effort when the plan is still partial by attempting to deal with conditions that are not ultimately established by the abstract step’s subplans.

Analogously, if the operator writer puts in too few preconditions or effects in the abstract action’s description, then the planner may be unable to prune out unworkable plans at an early stage, giving rise to less-efficient planning.

3.3 Writing Decomposition Operators

Decomposition operators specify subplans for a particular action operator. For any given abstract action there may be a number of decomposition operators, each specifying a different subplan for achieving the abstract action’s goals. Sample decomposition operators are shown in figures 2 and 3; a graphical representation of the operator from figure 3, showing the plan steps, links, etc, that are defined in the operator, is shown in figure 4. The parent step is shown above the subplan, and each box inside the shadowed region is a step of the subplan.

Subplans are always bounded by null begin and end steps, similar to the null begin and end steps that mark the initial and goal states of the plan as whole. These null steps are marked in figure 4 as B (begin) and E (end). These steps are automatically created by the planner and are not specified in the :steps slot of any decomposition operator. When the planner creates these null steps, it will automatically create lists of preconditions and effects for each of them. The begin step will have no preconditions, but its effects will be identical to those of the parent step’s preconditions. Similarly, the end step will have no effects, but its preconditions will be identical to the effects of the parent step (the only exception is when rewrite rules are used - see below). When the plan is instantiated, binding constraints are automatically generated that force the variables in the begin and end steps of each subplan to codeesignate with the corresponding variables in the parent step. In this manner, you can refer to the bindings of the parent step within the decomposition operator by referring to the
(define (decomposition describe)
  :parameters (?object)
  :constraints ((object ?object)
    (class ?object ?class)
    (has-parts ?object ?parts))
  :links ((step1 (know-about-hearer (class ?object ?class)) end)
    (forall ?part in ?parts
      (step2
        (know-about-hearer (has-part ?object ?part))
        end)))
  :steps ((step1 (describe-class ?object ?class))
    (forall ?part in ?parts
      (step2 (describe-has-part ?object ?part)))
    (forall ?part in ?parts
      (forall ?other-part in ?parts
        (know-about-hearer (has-part ?object ?part))))))

Figure 2: A Describe Decomposition Operator for Objects

corresponding variable in either the begin or end step.

To specify a subplan for an abstract action, you must specify the steps that are required to appear in the subplan as well as any causal links and ordering constraints between the steps that you decide are essential to the subplan. You need not specify every step that will ultimately appear in the subplan (see section 3.3.7 for a discussion of partiality in decomposition definitions). Specification of the components of a subplan is done using the various slots of the decomposition operator (see the example in figure 2). I'll describe each of these components in detail below.

This decomposition operator specifies one of a number of possible ways to de-
(define (decomposition describe)
  :parameters (?process)
  :constraints ((process ?process)
    (class ?process ?class)
    (has-steps ?process ?steps))
  :links ((step1 (know-about-hearer (class ?process ?class)) end)
    (forall ?step in ?steps
     (step2
      (know-about-hearer (has-step ?process ?step))
     end)))
  :steps ((step1 (describe-class ?process ?class))
    (forall ?step in ?steps
     (step2 (describe-has-step ?process ?step))))
  :orderings ((step1 step2))
  :rewrites (((know-about-hearer ?process)
    (forall ?step in ?steps
     (know-about-hearer (has-step ?process ?step)))))

Figure 3: A Describe Decomposition Operator for Processes

Figure 4: Graphical Representation of A Sample Decomposition Operator

scribe an object. The subplan for description that this operator specifies involves an arbitrary number of steps. These include one step responsible for describing the
object’s class and a number of additional steps, each responsible for describing one of the components of the object.

The causal links of the subplan indicate that the describe-class step is used to inform the hearer about the class of the object, and each of the describe-has-part steps in the subplan is used to inform the hearer about the respective component parts of the object. The rewrites of the operator indicate that these effects taken together imply that the hearer knows about the object being described.

Constraints on decomposition operators restrict their applicability to certain contexts and serve to bind variables needed by $\texttt{forall}$ constructs in the other slots of the operator. In this example, the variable $\texttt{parts}$ is bound in the constraints to a list of the object’s subparts and is referenced in the $\texttt{:links}$ and $\texttt{:steps}$ sections of the operator to create new links and steps for each member of this parts list.

The constraints in this example also limit the operator’s use to context where the speaker is describing an object that has a list of parts (its subcomponents). This operator would be inappropriate, for instance, when applied to the description of processes, since processes are composed of subprocesses rather than subparts. The operator shown in figure 3 can be used to describe processes, and its constraints are appropriately different.

### 3.3.1 Parameters

The parameter specification of a decomposition operator serves to identify the variables that codeignate with the parent step’s variables and that are used by the decomposition operator’s $\texttt{begin}$ and $\texttt{end}$ steps. The $\texttt{:parameter}$ slot of a decomposition operator must contain a list of variables equal in length to the parameter list of the corresponding action operator. Although any variable names may be used in the $\texttt{parameter}$ slot, each variable that appears will be bound to the value of the variable in the corresponding position in the parent step’s parameter list at the time that the decomposition operator is instantiated.

### 3.3.2 Constraints

Constraints on decomposition operators serve two purposes. First, they limit the applicability of a decomposition. Second, they serve to bind variables referenced in other parts of the decomposition operator. Like constraints on action operators, decomposition constraints are always checked against the initial state for satisfaction and must not be conditions that appear as effects of other action operators. Also
like constraints on action operators, all constraints on a decomposition operator are checked at the time the planner first attempts to add the decomposition to the plan. The structure of the specification for decomposition constraints is identical to that for action constraints (that is, they appear as a list forming an implicit conjunction).

To determine if a constraint or set of constraints can be satisfied from the initial state, one of two mechanisms can be used. First (and simplest), the developer can enumerate all the facts that are true in the initial state by listing them in the problem definition. Second, the developer can use a special user-defined function to satisfy the open condition. This function may be linked to a data base, a knowledge base or some other application-specific source for determining what conditions hold in the initial state. For more information about this user-defined function, see section 5.

Other sections of a decomposition operator may refer to the bindings of variables in order to determine the number of conditional components to add to a particular subplan, as well as to determine what bindings the variables in these conditional components should have. Conditional components are steps, causal links, rewrites or orderings that are included in the operator at run-time based on the constraints' bindings (similar to macro-expansion in programming languages).

Conditional component specifications typically have the form:

\[
\text{(forall } \text{?var1 in } \text{?var2}
\]
\[
\text{ (form-1)}
\]
\[
\text{ ...}
\]
\[
\text{(form-n))}
\]

Here \text{?var2} must be a variable bound to a list of \text{m} elements by the constraints in the decomposition. \text{?var1} must be a variable not referenced outside the scope of a conditional component specification in that decomposition operator (that is, it must not be a variable bound in the constraints).

At the time the decomposition operator is used to expand an abstract step, Longbow will create \text{m} different instances of the forms within the body of the \text{forall}, each with \text{?var1} bound to a unique element from the list bound to \text{?var2}. These \text{m} instances will be added to the subplan as if they had appeared explicitly in the original decomposition operator.

\footnote{see the EBNF definitions in appendix B for the specific syntax.}
3.3.3 Steps

Every step that is required to occur in a subplan must be identified in the decomposition operator. To identify a step, its description must appear in the :steps slot in the decomposition operator. We call these descriptions pseudo-step descriptions. A pseudo-step description includes a name that is unique with respect to the pseudo-step descriptions of that decomposition and will be used to reference that step in other parts of the decomposition operator. This name is a label, very much like a variable used to refer to whatever step will be instantiated in the actual plan data structure. This label is used within the decomposition operator to provide a reference to this step and has no meaning outside the context of the operator in which it appears.

In addition to the specification of the pseudo-step label, the description must identify the type of the step, along with the specifications for each of the step’s parameters. For instance, the first step description appearing in the operator in figure 2 is:

\[(\text{step1 (describe-class ?object ?class)})\]

Here step1 is the label of the step to be created, describe-class is the type of the step to be created, and ?object and ?class are the step’s parameters.

Steps may also be specified as conditional components. To do this, place the step descriptions for the conditionals within a forall construct, as in the forall taken from the operator in figure 2:

\[(\text{forall ?step in ?steps}
\quad (\text{step2 (describe-has-step ?process ?step)}))\]

Here one new step will be created for every element of the list bound to ?steps, and in each new step ?step will be bound to a unique member of this list.

There are two steps that do not appear in the :steps slot but are automatically created by the planner for every decomposition decomposition operator. One is the decomposition’s initial step and the other its final step. These steps are essentially null placeholder steps used by Longbow to bound the temporal duration of the subplan. In addition, they play an essential role in maintaining the soundness of a plan as it is constructed by relating those conditions used in the subplan to the conditions relevant to the parent step.
Longbow creates each of these two steps and provides them with customized pre-conditions and effects depending on the context in which they appear. The initial step will be created with empty preconditions and with effects identical in form to the parent step’s preconditions. In this manner, the effects of the initial step act like the effects of the initial step of the plan as a whole, establishing the relevant state of the world at the point in the plan where the subplan begins execution. The final step of the subplan will be created with empty effects and with preconditions identical in form to the parent step’s effects. In this manner, the effects of the parent step are mirrored as preconditions in the final step of the subplan. This will cause Longbow to plan for the satisfaction of any of these preconditions that are not explicitly satisfied by the :links of the operator. In this manner, Longbow can ensure that each of the effects of the parent step is established at some point prior to the final step of the subplan.

The initial and final steps are automatically assigned the names begin and end, respectively, and consequently no other step in the :steps slot may use either of these symbols as a name. Similarly, the act-types of the initial and final steps are designated as start and end, respectively and a similar restriction on act-types for the other steps in the decomposition holds. The parameters of the initial and final steps are precisely those specified in the :parameters slot of the decomposition operator. At the time that a parent step is expanded, the bindings of the parent step’s variables will be copied to the variables used by these placeholder steps in the new subplan. In this way, the existing binding context of the plan can be extended into the subplan being created by the use of the decomposition operator.

3.3.4 Rewrite Rules

Rewrite rules are useful for changing the vocabulary from one abstraction level to another. When the effects of a parent step establish some condition $P$, and the steps in the subplan effectively make $P$ true by establishing a set of other conditions $Q_1$ through $Q_n$, one can use a rewrite rule to connect the parent effect to the effects of the steps of the subplan by rewriting $P$ as the conjunction of the $Q_i$, as described below.

Rewrite rules operate on the preconditions of the final step of the plan (recall that the preconditions of this null final step are identical to the effects of the parent step). A rewrite rule singles out one of these preconditions and substitutes a set of other preconditions in its place. As a result, steps in the subplan can be connected to these new preconditions and no longer have to establish exactly the same effects as
those of the parent step. The connection between the new set of preconditions and
the old rewritten precondition is saved in the plan’s data structures, however, so that
reference can be made to the overall structure of the plan after it is produced.

To specify a rewrite rule, you first identify the precondition to be rewritten and
then the list of new preconditions to be substituted as an implicit conjunction in its
place. The descriptions of preconditions to be replaced must use the same condition
name and variables as described for the final step in the :steps section of the operator.
If the variables do not match, the planner will be unable to determine which of the
preconditions is to be rewritten and an error message will (or should, anyway) be
generated at the time that the domain’s set-up function is called.

These rewrite rules and the domain axioms in UCPOP 2.0 are similar to each
other in that both mechanisms replace preconditions of actions with new sets of
preconditions. They differ, however, in three important aspects. First, UCPOP’s
domain axioms may be applicable to the preconditions of any step in a plan. Rewrite
rules are only applicable to the preconditions of the final step of the subplan in which
the rules appear. Second, many domain axioms may be applicable to a particular
precondition, and UCPOP applies all of them, generating a new plan for each possible
application. Only one rewrite rule can be specified for any precondition of the final
step of a subplan. Finally, unlike rewrite rules, domain axioms are applied iteratively.
That is, domain axioms replace a precondition with a new set of preconditions, each
of which might subsequently be replaced by the application of another domain axiom.
Rewrite rules are not applied iteratively. Once a precondition has been rewritten, its
replacement conditions cannot be rewritten.

Rewrites may also be specified as conditional components. To do this, place the
rewrite for the conditional within a forall construct, as in the forall taken from
the operator in figure 2:

```
((know-about-hearer (class ?process ?class))
  (forall ?step in ?steps
    (know-about-hearer (has-step ?process ?step))))
```

Here Longbow will rewrite the end step precondition (know-about-hearer (class
?process ?class)) as \( m \) new clauses (know-about-hearer (has-step ?process
?step)), where the value of step has been uniquely replaced with one of the \( m \) values
found in the binding for ?steps.
3.3.5 Causal Links

When two steps in a decomposition are to be connected by a causal link, the link must be defined in the :links slot of the decomposition operator. To define a link, you must indicate the source and destination steps (by referring to the labels you gave them in the :steps slot) and the conditions from these steps that play a role in the link.\(^5\)

So, for instance, in the operator in figure 2, there is a causal link that runs from the step labeled \texttt{step1} to the end step of the subplan (the step labeled \texttt{end}). This link establishes precondition \texttt{(know-about-hearer (class ?object ?class))} for the \texttt{end} step. To describe this link in the :links field of the operator, we write:

\[(\texttt{step1 (know-about-hearer (class ?object ?class)) end})\]

Links may also be specified as conditional components. To do this, link descriptions are placed within a \texttt{forall} construct. The syntax and meaning of the link description differs depending on whether the source and destination steps are themselves conditional components. There are four cases:

1. **When source and destination are both conditional steps:**

\[(\texttt{forall ?action in ?actions}
\phantom{(step2}
\phantom{(know-about-hearer (has-step ?process ?action))}
\phantom{step5}))\]

For this example, let’s assume that there are \(m\) elements in the list bound to \texttt{?actions}. There must also be a conditional component specification for \texttt{step2} and \texttt{step5} that uses \texttt{actions}. Consequently there will also be \(m\) instances of \texttt{step2} and \texttt{step5}. The form above will create \(m\) new links, each link with a unique source from the instances of the \texttt{step2} steps and a unique destination from the \texttt{step5} steps.

2. **When only destination is a conditional step:**

\(^5\)Since there may be several preconditions that match a given effect of some destination step (as similarly, several effects may match a given precondition of a source step), it’s necessary to spell out the connections in the link definition by indicating both the source effect and the destination precondition.
(step2
  (forall ?action in ?actions
   (know-about-hearer (has-step ?process ?action))
   step5))

For this example, let’s assume that there are \( m \) elements in the list bound to ?actions. There must also be a conditional component specification for step5 that uses actions. Consequently there will also be \( m \) instances of step5. The form above will create \( m \) new links, each link with the same effect of step2 as its source and with a unique destination from the step5 steps.

3. When source is a conditional step and destination is a conditional rewrite of the end step:

(forall ?action in ?actions
 (step2
   (know-about-hearer (has-step ?process ?action))
   end))

For this example, let’s assume that there are \( m \) elements in the list bound to ?actions. There must also be a conditional component specification for step2 and a conditional rewrite specification in the end step that uses actions. Consequently there will also be \( m \) instances of step2 and \( m \) copies of the condition (know-about-hearer (has-step ?process ?action)) in the preconditions of the end step. The form above will will create \( m \) new links, each link with a unique source from the instances of the step2 steps and a unique destination from the \( m \) conditions created in the end step’s preconditions.

4. When source is not a conditional step and destination is a conditional rewrite of the end step:

(step2
  (forall ?action in ?actions
   (know-about-hearer (has-step ?process ?action))
   end))

For this example, let’s assume that there are \( m \) elements in the list bound to ?actions. There must also be a conditional rewrite specification in the end step
that uses actions. Consequently there will also be \( m \) copies of the condition
\( \text{know-about-hearer (has-step \ ?process \ ?actions)} \) in the preconditions
of the end step. The form above will create \( m \) new links, each link with the
same effect of step2 as its source and with a unique destination from the \( m \)
conditions created in the end step's preconditions.

It is not a requirement that all steps defined in a decomposition operator be con-
ected to other steps in the operator by causal links. When a step in a decomposition
operator is not connected to the end step of the subplan via some chain of causal
links, this step is called unused. Unused steps may be isolated steps that have no
incoming or outgoing causal links, or they may be connected to small collections of
steps causally connected to one another but with no chain of links eventually leading
to the final step of the subplan.

Note that while an unused step may be the temporally closest step to some step
with an open precondition in the subplan, it may not be the case that the planner
will always chose the unused step to act as the source for a causal link satisfying the
open condition in a completed plan. Individual search control may guide the planner
to add a new step or to select some other existing step in the plan as the causal
link source. If this occurs, the unused steps may appear as redundant or excess
steps in the final plan. If the operator writer doesn’t wish to give the planner the
flexibility to select alternative steps to act as causal link sources, then unused steps
in decomposition operators should be avoided.

3.3.6 Orderings

There are a number of ordering constraints that are implicit in the structure of a
subplan. For instance, all the steps except the begin step will occur after the begin
step and all the steps except the end step will occur before the end step. In addition,
any step that is the source of a causal link listed in the :links section of an operator
will occur before the step that is the destination of the link. These ordering constraints
are created automatically by the planner upon instantiating the subplan.

However, operator writers may wish to impose additional ordering constraints
on steps. To do this, you simply list the sequences of steps to be ordered in the
:orderings section of the decomposition operator. Each list in an :orderings slot of
a decomposition defines a linear order between all of the list’s elements, restricting
the relative order of the steps in the list to the order in which the steps occur in the
list itself. The first step in a list will be ordered to precede the second, the second
before the third, etc. As an example, we can see that the step1 step and the step2 step in the operator shown in figure 2 need not be ordered since they don’t interfere with one another’s effects or preconditions. This operator does order them explicitly, however. The :orderings slot indicates that the step1 step occurs before the step2 step since the list (step1 step2) appears in the slot.

When precisely one step label in an adjacent pair of steps from an ordering list is one that appears as a conditional component (i.e., it is defined in the :steps slot within a forall), an ordering will be generated between the step that is not the conditional component and each of the actual steps added by the conditional. When both steps in an adjacent pair from an ordering list are conditional components, a pair-wise ordering will be generated for every pairwise combination of steps, one from each set.

3.3.7 The Partiality of Decomposition Operators

The subplan specified in a decomposition operator need not be a complete plan, that is, it need not have every step, causal link, and ordering constraint fully specified. When decomposition operators only partially specify a subplan, those components of the subplan in the decomposition are instantiated into the plan, and the standard planning process is responsible for completing the remainder of the subplan in any way it sees fit.

The ability to describe partial subplans in decomposition operators gives the operator writer more flexibility. For instance, if the writer wants a precondition of a step in the subplan to be established, but doesn’t care how it gets established, the writer can leave out a step in the decomposition that established the precondition via a causal link. The precondition in question will be treated as an open condition once the subplan is added to the plan as a whole, and the planner will eventually select that condition as the next goal to satisfy.

This flexibility comes at some cost, however. The planner must work harder to complete a partial subplan than to complete one that is fully specified. When a decomposition operator leaves $m$ flaws in a subplan, the planner will create $m \times n$ new subgraphs in the plan space, assuming there are $n$ ways of addressing each flaw. This can give rise to an uncomfortably large number of new plans to explore at each decomposition in the plan search space. It is suggested that the operator writer take care to consider the trade-off between flexibility and search space size when writing operators for specific domains.
3.4 Defining a Set-Up Function for a Planning Problem

The set-up function that Longbow uses performs three important initialization tasks. First, it clears the global variables that store action templates, erasing any old operators left behind by previous problems. Second, it defines a problem function, a function that defines the action and decomposition operators for the current domain. Third, it defines the problem structure - the data structure that defines the initial and goal states and associates them with the planning function.

We show the EBNF form for the set-up function for planning problems in figure 5. Note that the asterisks that appear in the constant names *action-templates*, *decomp-templates* and *tests* are not the closure operators of EBNF forms but, rather, are a component of the Lisp token indicating a Lisp global variable. For EBNF definitions of action and decomposition operators, see appendix B.

3.4.1 Defining the Problem Structure

A problem structure contains a description of the initial and final states of a particular problem and a pointer to the problem's domain, the operators for the particular problem. The :domain slot holds the function that defines the operators to be used for this problem. The initial and goal states for a given problem are specified in two slots the problem structure definition: the :init and the :goal slots, respectively. Both the initial state and goal states are described by listing the positive conditions that hold in them. The initial state specification must describe every fact that is true in the world at the time the plan will begin. Longbow makes a closed-world assumption, that is, it assumes that every fact not listed explicitly in the initial state description is false. Consequently, only positive conditions can be listed in the initial state description.

The initial state may also be described programmatically. Section 5 describes the set-up required to allow a user-defined function to compute the initial state, both for preconditions and for constraints, as required during planning.

Like the initial state description, the goal state description is also a list of conditions; it is used to indicate what conditions must hold when the plan has completed. Unlike the initial state description, the goal state may list negative conditions as well as positive ones. Longbow requires that the goal state description must be consistent.

Both the initial state and goal state descriptions are encoded in the plan as conditions on dummy placeholder steps. A null step is automatically created whose effects are exactly those conditions in the initial state. This step is constrained to be the
Figure 5: EBNF Form for a Planning Problem

first step in the plan and no steps are allowed to be added before it. When some step’s precondition depends on a fact in the initial state, Longbow creates a causal link from this null initial step just like it would from any other. Similarly, a null final step is created whose preconditions describe the goal; in order to complete the plan, Longbow must close all the open preconditions of the null final step.
3.5 Syntax Checking

Longbow operator definitions are checked for validity at two points. First, Lisp enforces its own syntax requirements when instantiating the structures that hold the operator templates. If an operator definition violates the slot definitions for an action or decomposition structure, Lisp will break when executing the definition form; when this happens you will receive a Lisp-generated error message.

Second, syntactic and semantic constraint checking is done programmatically by the Longbow function test-ops. This function is automatically called every time a problem is run, immediately before the problem is handed over to the plan search routine. If you would like to turn off this automatic checking, set the variable *auto-test-ops* to nil

You can verify these constraints directly by calling test-ops yourself. Before test-ops can be called to check a set of operators, the operators must be assigned to the template variables and the problem-function for the problem must be defined. Once these conditions are established, call test-ops on the problem function.

If you are iteratively repairing operator definitions, make certain to re-load the operator definition file that you’re working on, re-run the set-up function, thereby re-initializing the operator templates and re-defining the problem function each time before running test-ops.

4 To Build a Plan

4.1 How Longbow Searches for Plans

When you define a Longbow planning problem you are defining a plan space - a space made up of all the possible plans that have the initial and goal states you’ve defined and are composed of steps made out of the operators in your problem definition. Longbow goes about searching this plan space for solutions to the problem you define.

Fortunately, this space of plans is organized in a way that makes searching through it a little more straightforward than a random search would be. Each node in this space is a (possibly partial) plan. The space forms a graph with a single root; this root node is the initial null plan that contains only the initial and final steps specified in the problem structure. Longbow starts its search by looking at just this node, incrementally constructing more of the space as it searches.

Longbow keeps a queue of unexpanded nodes in the plan graph (stored in the variable *q*); initially this queue contains only the graph’s root node. The queue is
• Create null plan with the initial and goal steps corresponding to the initial and goal steps of
the planning problem. Assign this plan to the variable *p*.
• Set the plan search queue variable *q* to the list containing only *p*.
• Set the flag DONE to nil and the counter SEARCH-COUNT to 0.
• While not DONE and SEARCH-COUNT < SEARCH-LIMIT do:
  • Use the function rank-plan to sort *q*.
  • Pick the best plan from *q* and assign it to *p*.
  • Remove *p* from *q*.
  • If *p* has no flaws, set DONE to T, else
    • Use the function get-flaw to pick a flaw f of *p*.
    • Use the function plan-refinements to create a new plan \( p_i \) for every refinement \( i \)
of f in *p*.
    • Add each \( p_i \) to *q*.
    • Increment SEARCH-COUNT.
• if DONE return *p* else
  • Report search limit exceeded.
  • Return FAIL.

Figure 6: Overview of Longbow Plan-Space Planning Algorithm

sorted based on the ranking assigned to each of its members by the ranking function
**rank-plan** (see section 4.2.1). The most preferred plan is taken off the queue, a
flaw of this plan is chosen to be repaired using the function get-flaw and the plan’s
children are created by repairing that flaw using the function plan-refinements (see
section 4.2.2). These children replace their parent in the queue. The process iterates
until either a solution is found or a search limit is exceeded.\(^6\)

\(^6\)On each iteration of the search algorithm, a counter is incremented – if this counter ever exceeds
the search limit stored in the variable *search-limit*, Longbow will halt and return nil. This
4.1.1 Plan Flaws

A partial Longbow plan will have any number of flaws. There are three types of flaws that Longbow plans may have. They may have open conditions, preconditions of steps that are not yet established by causal links. They may have unexpanded abstract steps, abstract steps that have not yet been expanded by the addition of subplans. And they may have unsafe causal links, links whose conditions are at risk of being undone (or threatened) by other steps. A plan’s flaws are all found in its flaws slot. We describe each of these types of flaw below:

- **Open Preconditions:** Conceptually, an open precondition is one that is not satisfied immediately prior to the execution of the step to which the condition belongs. Unless all the preconditions of a step are established before the plan executes, the step will not execute correctly. Open preconditions in Longbow are preconditions that do not have a causal link establishing them. Every open precondition in a plan is represented using an open data structure. When a new step is added to a Longbow plan, all of its preconditions are open.\(^7\)

- **Unexpanded Abstract Steps:** An unexpanded abstract step is an abstract step that does not yet have a subplan. An unexpanded abstract step in Longbow is one that has no decomposition links to any children steps. Every unexpanded abstract step in a plan appears in the flaws slot of the plan. Abstract steps are considered unexpanded immediately upon being added to a plan.

- **Unsafe Causal Links:** An unsafe causal link is one that is threatened by some step. A step threatens a causal link when it could possibly appear during the interval spanned by the link and one of the step’s effects can unify with the negation of the condition labeling the causal link. Every unsafe causal link in a plan is represented by an unsafe data structure in the flaws slot of the plan.

4.1.2 Repairing Plan Flaws

Longbow’s task is to construct a plan that is free of flaws. As described above, flaw repair is the basic technique used to search through the space of plans. Each type of search bound is used to keep Longbow from constructing an infinite plan in plan spaces where no solution exists.

\(^7\)If the new step is added as the result of a decomposition operator, it may immediately have its preconditions closed by causal links, but only if the links are specified in the :links slot of the decomposition operator.
flaw listed above has a well-defined set of techniques for repairing it. Longbow uses these techniques to generate children plans during its search. These techniques are listed here, categorized by the type of flaw they repair.

- **Open Conditions:** There are two basic methods for repairing open condition flaws. Both involve adding a causal link to the plan to establish the needed condition. The first method finds a step already in the plan that

  - may occur prior to the step with the open precondition, given the current ordering constraints and
  
  - has an effect that can unify with the open condition, given the plan’s current binding constraints.

Once such a step has been found, a causal link is created between the two steps, an ordering constraint is added to the plan constraints in order to force the source step to occur before the destination step and any binding constraints needed to make the effect of the source step unify with the open condition are added to the binding constraints of the plan.

The second technique is similar, but involves adding a new step to the plan. The action templates are searched for any operator whose effect could unify with the open condition. Any template that is found is used to instantiate a new step; this new step is added to the plan, a causal link is created and the appropriate binding and ordering constraints are also included.

During search, Longbow creates a new child plan for each way that can be used to establish the needed condition.

- **Unexpanded Abstract Steps:** An unexpanded abstract step flaw is repaired by the addition of a subplan below the unexpanded step. To add a subplan for an abstract step, a decomposition operator is chosen whose act-type is the same as the type of the unexpanded step. The decomposition operator specifies a small plan fragment, complete with its own set of new constraints. The steps specified in the operator, along with the other constraints it describes, are added to the constraints of the plan. Then decomposition links pointing from the parent step to the steps of the subplan are added to the plan’s set of decomposition links.

---

8Note that currently Longbow uses a standard unification algorithm to match conditions. For a discussion of the unification algorithm, see ??.
The process of expanding a single abstract step by adding its subplan is more complex than the other types of flaw repair performed by Longbow. Figure 7 gives a high-level description of the step decomposition process. During search, Longbow creates a new child plan for each decomposition operator that can be used to expand an unexpanded abstract step. Furthermore, when adding steps in a subplan, Longbow can choose to re-use steps already in the plan rather than instantiating new ones. Distinct child plans are created for each of these options.

- **Unsafe Causal Links**: A causal link is considered unsafe when some step in the plan might undo the link’s condition sometime during the interval spanned by the link. Steps like this are called threats to the link. To qualify as a threat to a causal link, a step must meet three conditions:

  1. The step must possibly occur prior to the destination step of the link.
  2. The step must possibly occur after the source step of the link.
  3. An effect of the step must be able to unify with the negation of the condition of the link.

Repairing an unsafe link flaw caused by a threatening step involves attacking one of these conditions. There are, consequently, three ways to repair an unsafe link:

  1. **Promotion**: Add an ordering constraint to the plan’s orderings that forces the threatening step to occur before the source step of the link.
  2. **Demotion**: Add an ordering constraint to the plan’s orderings that forces the threatening step to occur after the destination step of the link.
  3. **Separation**: Add binding constraints to the plan’s constraints that will keep the effects of the threatening step from unifying with the negation of the link’s condition.

---

9 This figure shows the process at a conceptual level. The actual code varies somewhat from the algorithm sketched here. Consult the file `code;expand.lisp` for the specific functions responsible for step expansion.

10 In this way, Longbow can create plans that are graph-structured rather than tree-structured. Graph-structured plans are ones in which one or more steps are shared between two distinct subplans.
Each one of these approaches results in the creation of one new child plan in
the plan space.

4.2 Controlling Longbow Search

A Longbow application developer can customize the Longbow search process in sev-
eral ways. The most straightforward approach is to change the ranking function
\texttt{rank-plan} or the flaw selection function \texttt{get-flaw}. Alternatively, one can change
the search function that Longbow uses. Several alternative search functions de-
defined for UCPOP (but customizable for use with Longbow) can be found in the
file \texttt{lb:code;interfac.lisp}.

4.2.1 Ranking Plans

Longbow uses a best-first search to search the space of plans. Before a plan is added
to the queue of unexplored nodes, it is given a non-negative integer ranking. This
ranking is performed by the function \texttt{rank-plan}. The higher the ranking assigned
to a plan, the worse the plan is considered to be. When \texttt{rank-plan} assigns a plan a
value of -9999, the plan is a complete solution to the planning problem and will be
returned as the first value of the user’s call to \texttt{find-plan}.

Ranking is useful for search efficiency. When new plans appear unpromising, they
are given a higher ranking than their more attractive relatives. As a result, they are
moved towards the rear end of the queue and not considered until all plans of lower
rank (higher preference) have been considered and have failed to produce a solution.

Ranking also can be used to enforce domain-dependent preferences for plan struc-
ture. For instance, if the system designer prefers solutions to a planning problem that
have steps of a certain type, the ranking function can be modified to give plans with
those types of steps a higher rating than plans without them. The designer must take
care, however, to make sure that all the different factors that go into ranking plans
interact correctly in all cases. If they do not, the planner may run very inefficiently
or may exceed its search bound, failing to find solutions.

A counter-intuitive aspect of ranking when building hierarchical plans is that one
often has to rank plans with more abstract steps over plans that may be closer to a
complete solution but that do not use as many abstract steps. This is due to the fact
that primitive steps may also establish the same conditions as abstract steps. For
some simple problems it may be easier to build a solution out of primitive steps than
Given a parent plan, an unexpanded parent step in that plan, and a decomposition operator for that parent step:

- Let NEW-PLANS be nil.
- Create a new set of bindings that extend the parent plan’s bindings by binding each of the variables specified in the final step of the decomposition to the corresponding variable in the parent step’s action operator.
- Using these bindings, satisfy the constraints of the decomposition operator, generating a list of new binding environments. For each binding environment:
  - Create the steps (both conditional and unconditional) specified by the operator.
  - Starting with the current environment, satisfy the constraints of each of the new steps being added to the subplan, generating a list of new binding environments. For each binding environment:
    1. Create a new plan data structure, copying the parent plan’s components into it.
    2. Order all new subplan steps to occur before the subplan’s end step and after the subplan’s begin step. Propagate every ordering constraint of the parent step to the appropriate bounding step of the subplan (i.e., its begin or end step). Order the new steps of the subplan according to any explicit orderings that appear in the decomposition operator.
    3. Rewrite the final step’s preconditions according to the current binding environment.
    4. Add all links specified in the decomposition operator to the new plan. Add all new bindings that arise from these new links to the plan. Add all orderings that arise from the addition of these new links to the plan.
    5. Add any new flaws, including new unexpanded abstract steps, new threats and new open preconditions, to the current plan.
    6. If the plan’s ordering and binding constraints are consistent, add the new plan to NEW-PLANS.
- Return NEW-PLANS.

**Figure 7: Overview of the Step Decomposition Process**

it is to build one out of a plan hierarchy. This may be a problem for applications that
require hierarchically structured plans.

4.2.2 Choosing Plan Flaws

Once the Longbow search function has selected the most promising plan to expand, one of the plan’s flaws must be selected to repair. The choice of the flaw interacts with the plan ranking function’s rank assignment to guide the search towards a plan solution. Different flaw selection strategies can be defined by the application developer that result in preferences for different type of plans. For instance, if unexpanded steps are always selected to be repaired before any other type of plan flaw, the planner will tend to complete each plan’s hierarchical structure before adding any causal links. Alternatively, if open preconditions are always selected before any other type of plan flaw, the planner will tend to construct a complete causal plan at one hierarchical level before detecting any threats or expanding any abstract steps.

The function `get-flaw` is responsible for picking a flaw to repair, and can be customized to suit the preferences of the developer. The default behavior for `get-flaw` returns threatened links before open preconditions before unexpanded steps.

4.3 A Simple Search Example

In order to illustrate plan space search, let’s consider the shop domain defined in `lb:ops;suite.lisp`, an extremely simple and artificial non-discourse planning problem. In the shop problem, the goal is to travel from home to the store. There are two ways to get to the store: you can walk there or you can take the bus. Two corresponding primitive actions are defined in this domain: `walk` and `take-bus`. Both walking and taking the bus have the single precondition that you’re at home and the single effect that you’re at the store.

To add a little abstraction to this domain, there’s a composite action called `travel` that takes us from home to the store. There are two decomposition operators for the `travel` action – one in which the only step in the subplan is a `take-bus` action and one in which the only step is a `walk` action.

Now let’s consider how Longbow will search the space of plans to find a solution. For this example, let’s assume a plan ranking function that prefers hierarchical plans

---

11 This is an artificial domain and is not meant to be illustrative of Longbow’s representational abilities. It is intentionally simplistic in order to demonstrate the complicated process of plan space search clearly. More realistic or better-designed sets of operators for this problem are not difficult to imagine, but would be harder to follow in the discussion below.
In this figure and the plan-space figures that follow, each plan in the plan space is represented as an oval node containing a rough diagram of the plan using smaller boxes and arcs to represent individual steps and links in the plan. Arcs connecting the oval plan nodes indicate refinement relationships between the plans. A node at the destination of an arc is the refinements of the node and the source end of the arc.

Figure 8: Initial Plan Space for Store Problem

to non-hierarchical plans and a search strategy that looks first at unexpanded abstract steps, then open preconditions and finally threats to causal links.

Figure 8 shows the plan space at the beginning of the search. The oval labeled P1 represents the null plan containing only the initial and final steps. The initial step has one effect, that of being at home, while the goal step has one precondition, that of begin at the store. The plan P1 has one flaw: the open precondition of the final step (AT-STORE). If we were to execute the form (get-flaw p1), Longbow would return the open flaw data structure corresponding to this condition.

As described above, there are two ways that Longbow can resolve an open condition flaw. One way is for Longbow to find a step already in the plan that can be used to establish the needed condition. To do this, it would first have to locate a step that comes before the step with the open condition and whose effect unifies with it. Then it would create a causal link from this step to the step with the open condition. Alternatively, Longbow can find an action operator template whose effect unifies with the open condition. Then it can create a new step from this template, add it to the plan before for the step with the open condition and create the corresponding causal link.

In this example, there is no step already in the plan that could serve as a source for a new causal link. Longbow does find three action templates, however, whose effects unify with the open: the travel operator, the take-bus operator and the walk operator. Consequently, Longbow creates three new plans – one for each of the ways of resolving the open condition flaw. These three new plans are shown as the leaf nodes labeled P2, P3 and P4 in figure 9.
At this point in the search, the search queue \( q \) contains \( P_2 \), \( P_3 \) and \( P_4 \). Because our ranking function prefers hierarchical plans, and because \( P_2 \) is the only plan with an abstract action in it, \( P_2 \) receives the highest ranking and so is chosen to be expanded next.

Plan \( P_2 \) has two flaws: the unexpanded \textit{travel} step and the \textit{travel} step’s open precondition (\textit{AT-HOME}). Because our search function chooses unexpanded step flaws to resolve before other types of flaws, Longbow chooses to address the unexpanded \textit{travel} step flaw next. To resolve an unexpanded step flaw, Longbow must find a decomposition operator for the act-type of the unexpanded step. In this example, there are two decomposition operators for the \textit{travel} step – one for walking and the other for taking the bus. Longbow creates a new plan node in the plan space for each of these alternatives. In figure 10, the plan labeled \( P_5 \) represents the expansion of the travel step by adding the \textit{take-bus} subplan, while the plan labeled \( P_6 \) represents the addition of the \textit{walk} subplan. After creating these plans, the parent plan \( P_2 \) is replaced in the search queue by the two plans \( P_5 \) and \( P_6 \). At this point, the queue contains plans \( P_3 \), \( P_4 \), \( P_5 \) and \( P_6 \). Since the ranking function prefers hierarchical plans to non-hierarchical ones, plans \( P_5 \) and \( P_6 \) are ranked above \( P_3 \) and \( P_4 \).

Plans \( P_5 \) and \( P_6 \) are structurally identical down to the act-types of their steps. They have the same number of actions, causal links and other plan constraints. They both have a single flaw (the open condition (\textit{AT-HOME}) of the \textit{travel} step for both plans). Their only difference is that \( P_6 \) has a \textit{walk} step where \( P_5 \) has a \textit{take-bus} step.
Due to their similarity, a typical plan ranking function might not distinguish between the two, giving them both the same ranking. If this happened, Longbow would simply take the first element of the queue. But imagine instead that we had specified some domain-specific information in the ranking function. Suppose we’re lazy and really don’t want to have to walk to the store if we don’t have to. Consequently, we’ve designed the plan ranking function so that, when all other factors are equal, the function will rank a plan to take a bus to the store higher than a plan to walk there.

As a result, plan P5 is given higher ranking than P6 at this point and is selected for expansion. As we described above, the only flaw that P5 has is the open condition \((\text{AT-HOME})\). There are no operator templates that have an effect that will unify with \((\text{AT-HOME})\), so Longbow cannot instantiate a new step to resolve this flaw. But Longbow can find a step already in the plan that will contribute the needed condition. Longbow creates the new plan P7 by adding a causal link from the \textit{initial state}\textsuperscript{1} to the \texttt{travel} step. P7 replaces its parent plan P5 in the search queue. At this point the search queue contains plans P7, P6, P4 and P3.

Plan P7 has been \textit{completed} – it has no more flaws and is a solution to the planning problem. So when the ranking function considers the search queue next, it marks P7 as complete and Longbow halts, returning P7 as its answer. The space that Longbow searched for this problem is shown in figure 11.

\textsuperscript{1}recall that both the initial and goal states are represented in Longbow as steps in the plan.
The entire plan space for this problem is shown in figure 12. By comparing this figure to figure 11, you can see that there are a number of plan nodes that were not explored by Longbow. By designing the ranking function differently, we could have taken a different path through the space, resulting in solutions with different structure being returned or resulting in a more efficient or less efficient search. For instance, if the ranking function had preferred plans with shallow hierarchical structure (or no hierarchical structure at all), Longbow would have expanded plans P3 or P4 before searching below P2 and consequently found P9 or P10 as solutions.

4.4 Examining Plan Spaces, Individual Plans and Plan Components

4.4.1 Examining the Plan Space

There are basically three ways that developers can examine the plan space of a planning problem. First, the developer can use one of the graphical user interfaces discussed further in section 6. This option may be the most straightforward, but developers' systems may lack support for the various interfaces provided with Longbow. In this case, the developer can use the plan creation functions themselves to move about the plan space in the lisp listener, starting at the root node and creating the tree of plans “on demand,” exploring those paths of interest to the particular problem. However, for all but the smallest of plan spaces this approach is likely to be rather tedious. Nevertheless, functions for performing this kind of plan space navigation are described in greater detail in section C. A more attractive alternative is to employ the suite of printing functions designed to display plan spaces and individual plans. It is this final approach that is described in this section.

Each time the planner is run, the plan space graph explored by the run is saved.\footnote{This space will almost certainly be less than the problem's plan space graph, since the whole plan space may exceed the search bound limit or the planner may simply be looking for the first complete solution rather than all solutions.} The root of this graph is held in the global variable *initial-plan*. To examine the entire plan space for a problem, the developer should call the function print-plan-space. This function accepts a number of keyword arguments that control the information it prints about the plan space (See appendix C for details). This function prints the plan space as a tree, indicating the relationships between parents and children plans in the space and displaying index numbers for each plan in the graph that can be used
to uniquely identify individual plans when calling plan inspection functions described below.

Developers should note that in order to make Longbow’s use of memory more efficient, some data structures are shared across plans in the search space when possible. For instance, the data structures representing the initial and goal steps are the same structures in every plan in a plan space. You should be wary of any information stored in a shared data structure that is modified by Longbow. It may be useful only in the context in which it is manipulated. For instance, pointers from a parent step to children steps in the parent’s subplan may be invalid when viewed in a plan where the parent has yet to be expanded.

4.4.2 Examining an Individual Plan

The ability to access the structure of a plan is useful both during the development of a set of operators for a given application and after an application’s operators have been debugged. As with the viewing of the plan space graph for a problem, there are several ways that individual plans can be inspected. Developers can always use the plan slot access functions described in appendix C; they can also use the suite of inspection and printing functions designed to display individual plans and plan components. It is this later approach that is described in this section.

In order to inspect an individual plan, developers must first use Longbow to create the plan structure. When the plan of interest is a solution to the current planning problem, this can be done by calling a plan search function like find-plan. When the plan of interest is not a solution, but rather some other plan node in the space of plans for a planning problem, the easiest way to obtain a pointer to the plan structure is by using a three-step process. First, create the plan space graph by calling a plan search function (again, one like find-plan) Then, determine the index number of the plan of interest by inspecting the nodes in the plan space using the function print-plan-space. Finally, obtain the plan structure by using the function get-plan-from-space.

Developers can see the structure of an individual plan by using the function print-plan. print-plan takes as an argument the plan of interest and a number of additional keyword arguments that specify the settings for various printing options. Alternatively, the function print-plan-from-space can be called when the ID number of the desired plan is known.

To obtain a list of the executable steps in a plan, i.e., those steps at the leaves of the plan structure, developers can use the functions execution and execution*. 

42
Various additional functions for accessing plan structure are discussed in appendix C.

4.5 Simple Template Functions for Generating Output

When Longbow is used to generate discourse plans, the executable steps of a solution plan are typically translated into some sequence of textual sentences realizing the communicative actions of these steps. This translation process may be a complex one, for instance, involving the generation of functional descriptions and the use of a system like FUF [2] to compute the text output. Or it may be more simple, for instance, using template-based techniques to translate each step data structure into an individual clause or sentence.

Longbow provides some support for a simple, template-based realization approach. Each action operator may use its \texttt{:template-fn} slot to specify a template function used to translate instances of the action into the desired output form. For instance, once a collection of executable step structures has been found, user-defined code may iterate over these steps, applying each step’s template function to the step itself. The specification of what internal step data structures are accessed and what text or other type of output is generated by the template functions is application-dependent and is the responsibility of the developer. An example of this type of translation can be found in the file \texttt{lb:ops;html.lisp}.

4.6 Debugging Your Longbow Domains

The approach you take to building a Longbow application is as much a matter of style as it is of software engineering. You can write all the operator definitions, describe a characteristic pair of initial and goal states and then debug the problem en masse. Or you can build the domain more incrementally, writing a single operator at a time and testing it using a planning problem that requires the minimal amount of new plan components. Because the plan spaces for even simple operator sets can be quite complex and because plan space size grows rapidly as a function of the number of operators in a domain, I recommended the incremental approach to constructing and debugging domains. Wherever along this scale your design choices fall, at some point you will still need to figure out why your operator sets cause Longbow not to do the right thing for a particular planning problem.

Debugging Longbow domains typically will involve the iteration through a three-stage process. First, the developer will need to examine the entire plan space to
locate points in the search process where Longbow does something unexpected (i.e., where Longbow creates an incorrect plan or fails to create a correct one). Then, the developer will need to examine the plan involved in the incorrect plan construction to determine why Longbow did what it did at that point in the search process. Finally, as a result of this investigation, the developer will likely be able to isolate errors in the relevant operators or constraint satisfaction methods, revise her domain specification and repair the mistakes.

The first step in this process is to examine the plan space to isolate where the planning error is occurring. This can be done by using the functions described in section 4.4.1 and appendix C. By looking at the structure of the space, developers interested in debugging a domain should be able to isolate points where Longbow has done the unexpected. As mentioned above, the two ways this may occur are:

- Some children plans for a given node may be incorrect. This will typically happen when operators are accidentally too liberal (e.g., when an operator’s constraints are not specific enough, when the constraints themselves are erroneously satisfied when they should not be).

- Some or all of the anticipated children plans for a given node are missing. This will typically happen when operators are accidentally too strict. For instance, when variable bindings are accidentally inconsistent or when an operator’s constraints erroneously fail.

In the first case, developers should identify the index number of the plan that is the parent of the incorrect plan or plans. A refinement operation on this parent plan created the incorrect children plans, and developers will be able to locate the specific error in their domains by examining the refinement performed on the parent.

In the second case, developers should identify the index number of the plan that would be the parent of the absent plan or plans. A failure to refine this parent plan correctly gave rise to the missing children plans, and by examining the refinement performed on the parent developers will be able to locate the specific error in their domains. The functions and the approach described in section 4.4.2 can be used to examine particular plan structure that will indicate more precisely where these errors might arise. There are three ways that domain operators can cause Longbow to do the “wrong” thing:

- The function find-plan can return nil when you expect it to find a plan. This occurs when your operators have errors that prevent Longbow from building the plan you expect.
• Longbow can exceed its search limit while executing `find-plan` in situations where you expect to find a plan. This can be caused by errors in your operators or by inappropriate rankings made by the rank function `rank-plan` guiding the search away from the solution.

• `find-plan` can return a plan other than the one you want. This can also be caused by errors in your operators or by inappropriate rankings made by the `rank-plan` rank function.

5 Connecting Longbow to a Knowledge Base

Longbow provides two methods for the programmatic satisfaction of conditions. The first, used for the satisfaction of preconditions checked against the initial state, is discussed in section 5.1. The second, used for the satisfaction of constraints (either action or decomposition operator constraints) is discussed in section 5.2.

5.1 Satisfying Preconditions Programmatically

As we described above, an open precondition is satisfied by establishing the appropriate causal link either from some preceding step or from the initial state to the step with the open precondition. When attempting to create a causal link from the initial state, Longbow has two methods available to determine what conditions hold there. First, it may check an explicitly enumerated list of conditions, as described in section 4.1.2. As an alternative, Longbow allows the user the option of routing all precondition queries to the initial state through a user-defined function that is responsible for computing the initial state on-demand as open preconditions are selected for satisfaction giving rise to causal link creation. When the variable `*longbow-precondition-kb-flag*` is set to `nil`, the establishment of links from the initial state works as described in section 4.1. When the variable `*longbow-precondition-kb-flag*` is non-nil, the creation of any causal link from the initial state is performed with the aid of the user-defined function `precondition-kb-query`, This function is described in the following section.

5.1.1 Specification for precondition-kb-query

Inputs:
- **clause** - This will be a single list whose first element is a relation name symbol and whose remaining elements will be parameters to the relation. All variables appearing in any of these parameters are free (unbound) in the current context.

- **binds** - a list of all variables appearing in the clause.

**What precondition-kb-query Should Compute:** The `precondition-kb-query` should consult a database, knowledge base or other user-defined source to see if there is a binding of the variables in the binds parameter that will instantiate the clause. The `precondition-kb-query` must compute *every* consistent set of such bindings.

**Output:** The meanings of the different values that can be returned by `precondition-kb-query` are:

- **nil** - this means that no consistent bindings could be found to make the clause true.

- (((t . t))) this means that no additional bindings were needed in order to make the clause true.

- \((b_1...b_n)\) a list of \(n\) elements, where each \(b_i\) corresponds to a unique set of bindings of the variables in the binds parameter that make the clause true in the user-defined database. Each element of \(b_i\) should be a dotted pair whose first element is a variable from the clause and whose second element is that variable’s new value.

### 5.2 Satisfying Constraints Programmatically

As we described in section 4.1, the satisfaction of constraints can be achieved by checking them against the initial state, where they must be enumerated explicitly. As an alternative, Longbow allows the user the option of routing all constraint satisfaction queries to one of two user-defined functions that are responsible for determining their validity and any subsequent bindings. When the variable `*longbow-constraint-kb-flag*` is set to `nil`, the establishment of constraint bindings works as described in section 4.1. When the variable `*longbow-constraint-kb-flag*` is non-nil, the satisfaction of constraints is performed with the aid of one of two user-defined functions: `constraint-kb-query` and `constraint-kb-query*`. These functions are described in the following section.
5.3 Satisfying Constraint Conditions in a Group

Some applications may require that an operator’s set of constraints be satisfied as a block (for instance, when domain-specific re-ordering of constraints is performed). Longbow provides the function `constraint-kb-query*` as an interface to a user-defined constraint satisfier that expects to deal with a collection of constraints rather than an individual constraint clause. For this function to be used to satisfy constraints, `*satisfy-decomp-constraints-as-a-block*` must be set to a non-nil value.

5.3.1 Specification for `constraint-kb-query*`

**Inputs:**

- **clauses** - This will be a list of lists whose elements are themselves clauses. Each clause will have as its first element a relation name symbol and have as its remaining elements the clause’s parameters. All variables appearing in any of these parameters are free (unbound) in the current context.

- **plan-binds** a list of n elements $(e_1\ldots e_n)$, where each $e_i$ corresponds to a unique binding environment for all of the variables in the current plan. Each element of $e_i$ should be a list of dotted pairs whose first element is a variable and whose second element is that variable’s new value.

- **plan** - the parent plan for the current refinement.

**What `constraint-kb-query*` Should Compute:** `constraint-kb-query*` should consult a database, knowledge base or other user-defined source to see if there is a binding for all the unbound variables in the clauses that will satisfy the clauses as a whole. `constraint-kb-query*` must compute *every* consistent set of such bindings.

**Output:** The meanings of the different values that can be returned by `constraint-kb-query*` are:

- **nil** - this means that no consistent bindings could be found to make the clauses true

- $(e_1\ldots e_n)$ a list of n elements, where each $e_i$ corresponds to a unique binding environment that makes the clauses true in the user-defined database. Each $e_i$ should correspond to one unique extension of one of the binding environments from the input `plan-binds`. 
5.3.2 Useful Functions for constraint-kb-query Writers

There are a number of functions that might prove useful for developers handling their own constraint satisfaction. These functions, listed below, range from ones dealing with variable binding to ones useful for breaking apart a condition into its constituent parts.

- Breaking apart conditions:
  - (get-relation condition): takes as input a condition condition and returns the relation for the condition. If the condition is a variable or an object constant, the function returns nil.
  - (get-args condition): takes as input a condition condition and returns a list of the arguments of the condition in the order in which they appear in the constraint specification. If the condition has no arguments, the function returns nil. If the condition is a variable or an object constant, the function also returns nil.

- Predicates for testing the type of condition components:
  - (constant? arg): Returns t if arg is a constant symbol and nil otherwise.
  - (variable? arg): Returns t if arg is a variable symbol and nil otherwise.
  - (condition? arg): Returns t if arg is a properly formed condition and nil otherwise.

- Functions for dealing with variable bindings:
  - (subst-bindings env condition-list): takes as input a binding environment env and a list condition-list. The function returns a list with every variable in condition-list replaced with the binding assigned to the variable in env. If no binding is assigned to a variable in condition-list, then the variable is not replaced.
  - (unify form1 form2 env): takes as input two forms form1 and form2 and a binding environment env. form1 and form2 may each be conditions, variables or constants. var and a binding environment env and
returns the value of the binding of \texttt{var} in \texttt{env}. If \texttt{var} is not bound in \texttt{env}, 
\texttt{bind-variable} returns the value bound to \texttt{var}.

- (bind-variable var env): takes as input a variable \texttt{var} and a binding environment \texttt{env} and returns the value of the binding of \texttt{var} in \texttt{env}. If \texttt{var} is not bound in \texttt{env}, \texttt{bind-variable} returns \texttt{var}.

### 5.4 Satisfying Constraint Conditions Individually

Some applications may require that action and decomposition constraints be passed to a user-defined constraint satisfier one clause at a time. Longbow provides the function \texttt{constraint-kb-query} for this use. For this function to be used to satisfy constraints, \texttt{*satisfy-decomp-constraints-as-a-block*} must be set to nil.

#### 5.4.1 Specification for \texttt{constraint-kb-query}

**Inputs:**

- \textit{clause} - This will be a single list whose first element is a relation name symbol and whose remaining elements will be parameters to the relation. All variables appearing in any of these parameters are free (unbound) in the current context.

- \textit{binds} - a list of all variables appearing in the clause.

**What \texttt{constraint-kb-query} Should Compute:** \texttt{constraint-kb-query} should consult a database, knowledge base or other user-defined source to see if there is a binding of the variables in the binds parameter that will instantiate the clause. \texttt{constraint-kb-query} must compute every consistent set of such bindings.

**Output:** The meanings of the different values that can be returned by \texttt{constraint-kb-query} are:

- \texttt{nil} - this means that no consistent bindings could be found to make the clause true

- \texttt{((t . t)))} this means that no additional bindings were needed in order to make the clause true.
• \((b_1...b_n)\) a list of \(n\) elements, where each \(b_i\) corresponds to a unique set of bindings of the variables in the \(\text{binds}\) parameter that make the clause true in the user-defined database. Each element of \(b_i\) should be a dotted pair whose first element is a variable from the clause and whose second element is that variable’s new value.

5.4.2 constraint-kb-query Example

As an example, suppose Longbow is attempting to satisfy the constraint \((\text{Mother } \text{mother } \text{child})\). The current plan bindings indicate that \(\text{mother}\) is already bound to the constant DOROTHY. To satisfy this constraint, Longbow first substitutes all the bound variables’ values into the clause (here creating the new clause \((\text{Mother DOROTHY child})\)) and then calls constraint-kb-query with the following form:

\[
\text{constraint-kb-query 'mother dorothy child '?child')}
\]

The function constraint-kb-query consults a user-defined knowledge base; let’s assume in this case that the only mother-and-child relations in the knowledge base are:

(Mother JALINA DOROTHY)
(Mother DOROTHY MIKE)
(Mother DOROTHY SANDY)
(Mother SANDY KRISTIN)
(Mother SANDY JENNIFER)

There are two sets of additional bindings that would make the query clause in this example true, one where \(\text{child}\) was bound to MIKE and one where \(\text{child}\) was bound to SANDY.

Consequently, constraint-kb-query should return

\[
'(\text{child MIKE}) (\text{child SANDY})
\]

6 Longbow’s Graphical User Interfaces

Because the graphical user interfaces to Longbow are not yet fully implemented, typical Longbow users currently run Longbow using a Lisp process connected to
a listener. However, the code for a CLIM 2.0 interface is included in the Longbow distribution. This code is supported and currently being extended; brave or desperate souls may find some solace in its use. We plan on developing a parallel interface in Garnet. The code for a simple Macintosh Common Lisp 2.0 graphical interface is provided with the Longbow system, but minimal support is available and future releases of the system are not guaranteed to remain compatible or to contain this code.

6.1 The CLIM 2.0 Interface

To load the CLIM 2.0 interface, add the following form to your lisp init file:

(push :clim-ui *features*)

This will cause the CLIM code to be loaded along with the Longbow system. To run the CLIM interface, execute the function

> (clim-gui)

Documentation describing the use of the CLIM interface will be available in The CLIM Interface Supplement to the Longbow Developer’s Guide. This document will be available on the web soon after the CLIM interface code is completed at the URL http://www.isp.pitt.edu/~young/longbow/docs.

6.2 The MCL 2.0 Interface

A Macintosh Common Lisp GUI interface is currently not supported. Although it will remain unsupported at least for the near-term, some MCL interface code is provided with the current release. Note: this code is subject to change or removal from future distributions without notice.

7 Acknowledgments

Support for the development of Longbow has been provided by the Office of Naval Research, Cognitive and Neural Sciences Division (Grant Number N00 014-91-J-1694)
and from the DoD FY92 Augmentation of Awards for Science and Engineering Research (ASSERT).

We are thankful to Johanna D. Moore, Martha Pollack, Giuseppe Carenini, Maria Gordin, Vibhu Mittal, Massimo Paolucci, Barbara Di Eugenio and Richmond Thomson for many helpful discussions during the design of the Longbow planner. A debt of gratitude is also owed to the alpha testers of the preliminary Longbow code, including Giuseppe Carenini, Mark Derthick, Barbara Di Eugenio, Michael Glass, Pam Jordan, Stefan Kerpedjiev, Vibhu Mittal, Carolyn Penstein Rosé and Steven Roth. Any remaining problems with the system are a result of my errors and not of theirs. Additional thanks to Vibhu Mittal for the code on which early versions of the CLIM 2.0 interface was based.

Thanks also to Martha Pollack for her collaboration during the development of the DPOCI planning algorithm. The Longbow source code makes extensive use of the UCPOP 2.0 code developed by Tony Barrett, J Scott Penberthy, Stephen Soderland and Daniel Weld at the University of Washington. We are exceedingly thankful for the use of their source code. The user should attribute all errors in coding or design to the Longbow code and not to the UCPOP implementation.

8 Disclaimer

All rights reserved. Use of the Longbow software is permitted for non-commercial research purposes, and it may be copied only for that use. The Longbow software is made available as is, and neither the author, the University of Pittsburgh nor the Office of Naval Research make any warranty about the software or its performance. In addition, no commitment of support for the software or its derivatives has been made.

The Longbow system was built using the source code for UCPOP 2.0, © University of Washington, 1990, 1991, 1992. UCPOP 2.0 was written by Tony Barrett, J Scott Penberthy, Stephen Soderland and Daniel Weld. We make no warranties with respect to the performance of the UCPOP code distributed with the Longbow system.

9 Cover Art

The cover art is an illustration taken from the mid-fourteenth century Lutrell Psalter. It shows English yeomen at Sunday longbow practice.
A Predefined Planning Problems

A number of planning problems have been defined in the file `lb:ops;suite.lisp`. Each of these domains are described below.

**test-describe3**

The `test-describe3` domain is an example of the use of Longbow in a simple discourse domain. `test-describe3` contains operators to describe an entity, either an object with sub-parts or an activity with sub-processes.

**store**

The `store` domain is used in the plan-space search example in section 4.3. The domain is non-linguistic and extremely simple.

B Longbow Operator Syntax

This section describes the extended BNF form for the syntax of action and decomposition operators. Additional constraints beyond those expressed in the EBNF forms are listed in section B.2.

**B.1 Operator Syntax in EBNF Form**

```
<action-operator> ::= (define (action <operator name>))
    [:description <string>]
    :parameters (<variable name>*)
    :primitive <boolean>
    [:constraints (<clause>*)]
    [:precondition (<clause>*)]
    [:effect (<clause>*)]

<clause> ::= (<predicate name> <term>*)
```
<term> ::= <clause>
<term> ::= <constant name>
<term> ::= <variable name>
<term> ::= <list variable name>

<operator name> ::= <act-type>

<constant name> ::= <symbol>
<predicate name> ::= <symbol>
<variable name> ::= ?<symbol>
<list variable name> ::= ?<symbol>

<string> ::= "<character>"'

<symbol> ::= <character>+ 

<character> ::= a..z
<character> ::= A..Z
<character> ::= 0..9
<character> ::= _
<character> ::= -

<decomposition-operator> ::= (define (decomposition <operator name>))
  [:description <string>]
  [:parameters (<variable name>*))
  [:steps <pseudo-step identifier list>
    [:links (<link identifier>*))]
  [:constraints (<clause>*))
  [:rewrites (<rewrite rule>*))
  [:orderings (<pseudo-step list>*))]

<pseudo-step identifier list> ::= (<pseudo-step identifier>*
  <start-step identifier>
<pseudo-step identifier>*
<finish-step identifier>
<pseudo-step identifier>*)

<pseudo-step identifier list> ::= (<pseudo-step identifier>*
<finish-step identifier>
<pseudo-step identifier>*
<start-step identifier>
<pseudo-step identifier>*)

<start-step identifier> ::= (<pseudo-step-name> (start <variable name>*))
<finish-step identifier> ::= (<pseudo-step-name> (finish <variable name>*))

<pseudo-step> ::= (<pseudo-step-name> (<act-type> <variable name>*))
<pseudo-step identifier> ::= <pseudo-step>
<pseudo-step identifier> ::= (forall <variable name> in <list variable name>
<pseudo-step>+)

<pseudo-step> ::= (<pseudo-step-name> (<act-type> <variable name>*))

<pseudo-step-name> ::= <symbol>
<act-type> ::= <symbol>

<link identifier> ::= (<pseudo-step name> <clause>
<pseudo-step name>)

<link identifier> ::= (forall <variable name> in <list variable name>
(<pseudo-step name> <clause>
<pseudo-step name>))

<link identifier> ::= (forall <variable name> in <list variable name>
<clause>
<pseudo-step name>))

<rewrite rule> ::= (<original clause> (<replacement conjunct>*))
<original clause> ::= <clause>

<replacement conjunct> ::= <clause>

<replacement conjunct> ::= (forall <variable name> in <list variable name> <clause>)

<pseudo-step pair> ::= (<symbol> <symbol>)

<pseudo-step list> ::= (<symbol> <symbol> <symbol>*)

B.2 Additional Constraints on Operators

There are a number of additional constraints on the structure of action and decomposition operators beyond those expressed in the previous section. These constraints are listed below.

B.2.1 Action Operators

1. No condition in a precondition, effect or constraint may contain a relation named \texttt{forall}. This token is reserved for the conditional constructs of decomposition operators.

2. Every action operator name must be unique.

3. Any variable that appears in a parameter list of an action operator must appear in that parameter list only once.

4. Any variable that appears in a precondition, effect or constraint of an action operator must appear in the parameter list of that action operator.

5. Any variable that appears in a parameter list of an action operator must appear in a precondition, effect or constraint of that action operator.

B.2.2 Decomposition Operators

1. Every decomposition operator name must be the same as some action operator name.
2. Within a single decomposition, the set of /variable name/ variables and the set of /list variable name/ variables must be disjoint. That is, /list variable name/ variables must appear only as terms in constraint clauses or as the list-bound variables in the forall clauses of conditional components. In contrast, /variable name/ variables must not occur as the list-bound variables in the forall clauses of conditional components.

3. Every pseudo-step identifier in a decomposition operator must be unique within that operator.

4. The two pseudo-step names appearing in a link identifier of a decomposition operator must both appear as pseudo-steps in the :steps field of that decomposition operator.

5. The predicate name in the first condition of a link identifier must match the predicate name of the identifier’s second condition.

6. Every decomposition operator must contain one and only one pseudo-step whose act-type is start. This step will be the initial step of the subplan and will automatically be ordered before all other pseudo-steps in the subplan.

7. Every decomposition operator must contain one and only one pseudo-step whose act-type is finish. This step will be the final step of the subplan and will automatically be ordered after all other pseudo-steps in the subplan.

8. The parameter variables for the initial and final steps of a decomposition must match each other, and the number of parameter variables for each step must also match the number of parameter variables of the action operator that the decomposition operator decomposes.

9. The orderings of a decomposition operator must be internally consistent. Furthermore, the orderings must be consistent with the implicit orderings that arise from the causal links specified in the links of the decomposition and must also be consistent with the constraint that restricts the begin step to occur before all other steps and the end step to occur after all other steps in the decomposition.

10. Every effect of a begin step must be traced via causal links to some precondition of the end step.
C Functions for Interacting with Longbow

While every Longbow function is useful in its own way, the following functions will be of particular use to you when integrating Longbow with other systems or when debugging your planning problems.

(compile-longbow) Defined in file loader.lisp

This function compiles all the source code files needed to run Longbow. It then loads them into the current lisp image by calling load-longbow.

(constraint-kb-query var binds) Defined in file dconst.lisp

(continue-finding-plans) Defined in file main1.lisp

This function can be called to continue searching for another solution to a current planning problem when search has already found at least one solution. This function is highly context-dependent, however, and assumes that all global variables and planner state set by the previously run planning function (either find-plan for new searches or continue-finding-plans for a continuation of the current search) remain unaltered.

This function returns the next solution plan that the plan search function encounters. It modifies the planner’s global variables accordingly.

(developer-mode) Defined in file interfac.lisp

This function switches the code mode to developer. When running in developer mode, Longbow will print out useful trace messages at important points during the planning process to act as diagnostics while debugging an operator set. Longbow will also save some useful debugging information in its plan structures when running in developer mode that otherwise might not be retained. To switch off the features present in developer mode, change into production mode using the function production-mode (see below).

(execution plan :include-dummies :include-unused) Defined in file inspect.lisp
execution takes as input a plan and returns an ordered list of step data structures corresponding to one possible ordering of the plan’s executable (primitive) steps. Note that execution picks an arbitrary ordering consistent with the ordering constraints of plan. There may be many different orderings possible. The user should not assume that the ordering returned by execution represents the only ordering possible. A user-defined ordering may be more appropriate for a specific application, in which case the user is free to define their own analog to the execution function.

- :include-unused: A boolean. When t, unused steps that appear in the plan will be included in the list of steps returned. When nil, these steps will not be included in the output. The default is nil.

  Note: currently the :include-unused keyword is ignored. Used and unused plan steps are included in output.

- :include-dummies: A boolean. When t, the null begin and end steps of subplans will be included in the list of steps returned. This is useful when plans may interleave subplans as a way to determine when interleaved subplans begin and end. When nil, these steps will not be included in the output. The default is nil.

(execution* plan :include-dummies :include-unused) Defined in file inspect.lisp

execution* takes as input a plan and returns an ordered list of step descriptions corresponding to one possible ordering of the plan’s executable (primitive) steps. This function is analogous to the execution function but rather than returning step data structures it returns a list of more readable forms (i.e., it returns each step’s p-step-action description with all variables replaced by their binding values). Note that execution* picks an arbitrary ordering consistent with the ordering constraints of plan. There may be many different orderings possible. The user should not assume that the ordering returned by execution* represents the only ordering possible. A user-defined ordering may be more appropriate for a specific application, in which case the user is free to define their own analog to the execution* function.

- :include-unused: A boolean. When t, unused steps that appear in the plan will be included in the list of steps returned. When nil, these steps will not be included in the output. The default is nil.
Note: currently the :include-unused keyword is ignored. Used and unused plan steps are included in output.

- :include-dummies: A boolean. When true, the null begin and end steps of subplans will be included in the list of steps returned. This is useful when plans may interleave subplans as a way to determine when interleaved subplans begin and end. When nil, these steps will not be included in the output. The default is nil.

(find-plan problem :search) Defined in file main1.lisp

- :search: A keyword, one of :exhaustive or :first. When :search is set to :first, Longbow will halt when the first successful plan solution to problem is encountered. When set to :exhaustive, Longbow will search the entire plan space (within the bound set by the value of *search-limit*) and collect all successful plan solutions. The default value is :first.

find-plan is the main listener-based interface function to Longbow. find-plan takes a problem name as input and searches the problem's plan space for a solution plan. If find-plan can find no plan in the space, or if the search limit is exceeded before Longbow halts, find-plan will return nil (in the later case, an error message indicating that the search limit was exceeded will also be printed). If find-plan finds a plan and the :search keyword is set to :first, find-plan will return two values. The first value will be the solution plan. The second value will be a data structure storing statistics about the search that Longbow performed to find the solution. If find-plan finds a plan and the :search keyword is set to :exhaustive, find-plan will a list containing all the plan solutions in the searched plan space.

As a result of calling find-plan, several global variables will be set. The global *p* will be set to the last plan explored. When find-plan succeeds in finding a solution to the planning problem and the :search keyword is set to :first, *p* will be bound to the solution plan that find-plan returns. The global *initial-plan* will be bound to the null plan that serves as the root node for the most recent plan search's search space. Finally, *q* will be set to the queue of unexplored plans in the plan space at the time that find-plan halted.
(get-children-steps step plan) Defined in file inspect.lisp

This function takes a step step and a plan plan and returns a list of step IDs corresponding the children steps of step in plan. If step is not a step in plan, or if step is a primitive step or is unexpanded in plan, then get-children-steps will return nil.

(get-children-steps* step) Defined in file inspect.lisp

This function is identical to get-children-steps except it operates on the plan stored in the global variable *p*. This is useful when referring to steps in the plan previously returned by a plan search function that sets the value of *p* appropriately (for instance, find-plan).

(get-decomposition step plan) Defined in file inspect.lisp

This function takes a step step and a plan plan and returns the decomposition schema used to expand step in plan. If step is not a step in plan, or if step is a primitive step or is unexpanded in plan, then get-decomposition will return nil.

(get-decomposition* step) Defined in file inspect.lisp

This function is identical to get-decomposition except it operates on the plan stored in the global variable *p*. This is useful when referring to steps in the plan previously returned by a plan search function that sets the value of *p* appropriately (for instance, find-plan).

(get-flaw plan) Defined in file main1.lisp

plan is any plan structure. get-flaw returns the flaw from plan that Longbow refines to create the children nodes of plan in the plan space. When get-flaw is nil, plan is complete.

(get-plan-from-space id) Defined in file inspect.lisp
id is an integer plan ID. get-plan-from-space returns the plan with the current plan ID from the current plan-space graph. When id is not the ID of any plan in the current plan-space graph, get-plan-from-space returns nil.

(1b-version ) Defined in file version.lisp

This function prints out the version information for the version of Longbow currently in use.

(1b-version-notes ) Defined in file version.lisp

This function prints out any comments associated with the current version. Typically, these comments describe recent changes between versions, minor implementation notes or other version-related information.

(load-longbow ) Defined in file loader.lisp

This function loads all Longbow code need to execute the Longbow planner.

(plan-binds plan) Defined in file struct.lisp

Returns the binding environment for plan.

(plan-children plan) Defined in file struct.lisp

plan is any plan structure. plan-children returns a list of plan structures that are the immediate descendants of plan in the plan space searched so far. When plan is a leaf node in the plan space searched so far, plan-children will return nil. plan does not have to be a complete or inconsistent plan in order for plan-children to return nil; the function may also return nil when plan is an unexpanded node in the search space.

(plan-flaws plan flaw) Defined in file struct.lisp

plan is any plan structure. plan-flaws returns a list of the flaws of plan. This list will contain OPENC structures (corresponding to open condition flaws), UNSAFE structures (corresponding to threatened causal links) and P-STEP structures (corresponding to unexpanded abstract steps). When plan-flaws returns nil plan is complete.
(plan-parent plan)          Defined in file struct.lisp

plan is any plan structure. plan-parent returns the plan structure that is the
parent of plan in the current plan space. plan-parent will return nil when
plan is the initial null plan for the planning problem.

(plan-reason plan)          Defined in file struct.lisp

(plan-top-level-steps plan) Defined in file main1.lisp

plan is any plan structure. plan-top-level-steps returns a list of step data
structures that are the top level steps in plan. That is, the steps returned are
exactly those steps of plan that are not part of a subplan of some other step in
plan.

(plan-refinements plan)     Defined in file grapher.lisp

plan is any plan structure. plan-refinements returns a list of new plans that
are the children of plan in the plan space. When plan-refinements returns nil,
plan is either inconsistent or is a complete plan. Note that plan-refinements
re-computes the children plans for plan rather than finding the corresponding
plans in the current plan space graph.

(print-plan-from-space
  num :stream :indent :right-margin :links-in
  :links-out :verbose :open-conditions) Defined in file inspect.lisp

print-plan-from-space takes as an argument a plan’s integer ID number and
prints out a text representation of the plan’s structure. The output of this
function is identical to that of print-plan.

- :stream: The stream on which the output of the function should be
  printed. The default value is t.
- :indent: An integer value indicating how far from the left margin to begin
  indentation when printing. The default value is 0.
• :verbose: A boolean. When t, the settings for the :reason and :flaw arguments will be overwritten to t. When nil, the values of these arguments will be used. The default is nil.

• :links-in: A boolean. When t the incoming causal links to the step will be printed along with the step description. When nil they will not be. The default is nil.

• :links-out: A boolean. When t the outgoing causal links from the step will be printed along with the step description. When nil they will not be. The default is nil.

• :open-conditions: A boolean. When t the open preconditions of the step will be printed along with the step description. When nil they will not be. The default is nil.

• :right-margin: An integer value indicating the number of characters from the left margin to the right margin in the output stream. The default is 80, but this value is currently ignored.

(print-plan-reason plan stream) Defined in file struct.lisp

print-plan-reason prints out a textual description of the reason that plan was created. This corresponds to the refinement of plan’s parent plan. stream is an optional argument indicating the stream to be used for output printing. Its default is t.


print-plan-space prints out a textual representation of the tree representing the current plan space graph.

• :stream: The stream on which the output of the function should be printed. The default value is t.

• :indent: An integer value indicating how far from the left margin to begin indentation when printing. The default value is 0.

• :counter: An integer value indicating which plan in array to use as the root for the function’s output.
- **:verbose:** A boolean. When `t`, the settings for the :reason and :flaw arguments will be overwritten to `t`. When `nil`, the values of these arguments will be used. The default is `nil`.
- **:reason:** A boolean. When `t` the reason the plan was created will be printed along with the plan description. When `nil` it will not be. The default is `nil`.
- **:flaw:** A boolean. When `t` the flaw in the plan that will be resolved next will be printed along with the plan description. When `nil` it will not be. The default is `nil`.
- **:right-margin:** An integer value indicating the number of characters from the left margin to the right margin in the output stream. The default is `80`, but this value is currently ignored.


Defined in file \texttt{inspect.lisp}

\texttt{print-plan} takes as an argument a plan and prints out a textual representation of the tree representing the plan defined by the plan’s steps, decompositions, links etc.

- **:stream:** The stream on which the output of the function should be printed. The default value is `t`.
- **:indent:** An integer value indicating how far from the left margin to begin indentation when printing. The default value is `0`.
- **:verbose:** A boolean. When `t`, the settings for the :reason and :flaw arguments will be overwritten to `t`. When `nil`, the values of these arguments will be used. The default is `nil`.
- **:right-margin:** An integer value indicating the number of characters from the left margin to the right margin in the output stream. The default is `80`, but this value is currently ignored.
- **:links-in:** A boolean. When `t` the incoming causal links to the step will be printed along with the step description. When `nil` they will not be. The default is `nil`. 

65
• :links-out: A boolean. When \texttt{t} the outgoing causal links from the step will be printed along with the step description. When \texttt{nil} they will not be. The default is \texttt{nil}.

• :open-conditions: A boolean. When \texttt{t} the open preconditions of the step will be printed along with the step description. When \texttt{nil} they will not be. The default is \texttt{nil}.

\begin{verbatim}
(production-mode )
\end{verbatim}

Defined in file \texttt{interfac.lisp}

This function switches the code mode to production. When running in production mode, Longbow does not print out the diagnostic trace messages printed during search when running in developer mode. To switch on the features present in developer mode, change into developer mode using the function \texttt{developer-mode} (see above).

\begin{verbatim}
(rank-plan plan)
\end{verbatim}

Defined in file \texttt{main1.lisp}

plan is any plan data structure. This function returns an integer value resenting the ranking of plan. When plan is complete, \texttt{rank-plan} will return -9999. Anytime plan is not complete, \texttt{rank-plan} should return a positive integer. The greater the integer, the worse the ranking of plan.

\begin{verbatim}
(test-ops problem-fn)
\end{verbatim}

Defined in file \texttt{opscheck.lisp}

problem-fn is a valid problem function. \texttt{test-ops} tests the syntax of the operators defined in the problem specified by problem-fn. \texttt{test-ops} prints out a sequence of error and warning messages to standard output indicating the source of any syntax errors it encounters. Note that when the global variable *auto-test-ops* is set to \texttt{t}, \texttt{test-ops} is called automatically each time a problem is run.
D Switc hes and Other Global Variables

D.1 Switc hes

D.1.1 Connection to the constraint-ko-query Function

*longbow-precondition-ko-flag*  Switch
When nil, Longbow uses the UCPOP mechanism for establishing causal links from the initial state. Otherwise, the precondition-ko-query function is accessed.

*longbow-constraint-ko-flag*  Switch
When nil, Longbow uses the UCPOP mechanism for establishing constraints from the initial state. Otherwise, one of constraint-ko-query or constraint-ko-query is accessed.

*satisfy-decomp-constraints-as-a-block*  Switch
When nil, Longbow passes each decomposition constraint to the constraint satisfaction mechanism individually in the order in which the constraints appear in a given decomposition operator. Otherwise Longbow will pass the entire block of constraints at once. Passing an entire block of constraints to the constraint satisfaction method is useful, for instance, when domain-specific reordering of constraints prior to their satisfaction check is required by a particular application.

D.1.2 Syntax Checking

*auto-test-ops*  Global Variable
When nil, Longbow will do no automatic syntax checking of operators. Otherwise, Longbow verifies the syntax and semantics constraints on operators each time the planner is called on a domain.

D.2 Global Variables

D.2.1 Template-Storing Variables

*action-templates*  Global Variable
Holds the definitions for the action operators.
*decomp-templates* 

Global Variable

Holds the definitions for the decomposition operators.

D.2.2 Plan-Space Bounds

*search-limit* 

Global Variable

Holds an integer value that limits the number of plan refinements done during any one search. When the number of refinements exceeds the value of *search-limit* during search, Longbow will halt, return nil and report that the search limit was exceeded.

D.2.3 Search State Variables

*q* 

Global Variable

Holds the current search queue for Longbow. This list corresponds to the unexpanded fringe nodes in the plan search space. Each element in *q* is a dotted pair whose first element is the numeric ranking of the plan and whose second element is the plan itself.

*p* 

Global Variable

*p* holds the plan that Longbow is currently operating on. After Longbow finds a solution and halts, *p* remains bound to the solution plan.

When using the CLIM interface, *p* can be set to a particular plan by displaying that plan in the plan display window. The plan that *p* is bound to can be displayed in the plan display window by selecting the command Show Current Plan.

*initial-plan* 

Global Variable

*initial-plan* holds the null plan that Longbow creates from the initial and final states specified in the planning problem structure. This plan is the root node of the search space and is used as the initial node for the search process.

D.2.4 Graphical User Interface Variables

*dpocl-dont-draw-null-steps* 

Global Variable

When nil, The Macintosh UI will draw all null steps when displaying a plan. Otherwise, null steps and the arcs leading into and out of them, will not appear.
*dpocl-draw-causal-links*  
Global Variable  
When nil, the Macintosh UI will not draw causal links when displaying a plan. otherwise, all causal links will be drawn.

*clink-threatened-color*  
Global Variable  
For the various user interfaces, contains the color value for drawing threatened causal links.

*decomp-link-color*  
Global Variable  
For the various user interfaces, contains the color value for drawing decomposition links.

*ok-causal-link-color*  
Global Variable

*null-step-color*  
Global Variable

*primitive-step-color*  
Global Variable

*unexpanded-step-color*  
Global Variable

*interior-node-color*  
Global Variable

E  A Brief Guide to Longbow File Structure

This section describes the contents of the directories contained in the current Longbow distribution.

E.1  lb: Directory

The lb: directory is the top-level directory for all Longbow code.

lb-init.lisp  
Lisp Source  
Code to be added to user’s lisp init files.
E.2  lb:docs Directory

This directory contains all documentation for the Longbow system, including research papers describing Longbow and DPOCL and manuals and papers related to UCPOP.

lb-cover.ps  
A cover page for the manual. Printing this page is optional.

lb-manual.ps  
This manual [7].

dpocl-nlgw94.ps  

dpocl-cogsci94.ps  
“Towards a principled representation for discourse plans,” a paper describing the intentional and informational discourse structure created by the DPOCL (and the Longbow) planner. From the Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society [9].

dpocl-aips94.ps  

ucpop-kr92.ps  

ucpop-manual.ps  
The UCPOP 2.0 User’s Manual [1].

E.3  lb:mac-ui Directory

This directory contains GUI code for the Macintosh Common Lisp 2.0 Longbow implementation. The Macintosh Common Lisp GUI interface is currently not supported.
Note: the MCL 2.0 interface code is subject to change or removal from future distributions without notice.

E.4  lb:clim-ui Directory
This directory contains the code implementing the CLIM. GUI interface for Longbow.

clim20.lisp  
Lisp Source  
Code for the CLIM 2.0 Graphical User Interface.

E.5  lb:contrib Directory
This directory is currently empty. All sample code and operator sets contributed by users will be stored here.

E.6  lb:code Directory
This directory contains the main body of code for Longbow.

const.lisp  
Lisp Source  
Code implementing action operator constraints in Longbow.

dconst.lisp  
Lisp Source  
Code implementing decomposition operator constraints in Longbow.

expand.lisp  
Lisp Source  
Code implementing decomposition of steps into their subplans.

grapher.lisp  
Lisp Source  

initkb.lisp  
Lisp Source  
Code to allow Longbow connection to an external knowledge base. See section 5 for details.

interfac.lisp  
Lisp Source  
Interface code to call the planning routines. Contains various control strategies for UCPOP.
loader.lisp
  Code that loads and compiles the Longbow system.

main1.lisp
  This file is one of two files containing the main code for Longbow. main2.lisp
  is the other.

main2.lisp
  Also contains main planner code.

opscheck.lisp
  Code for checking the validity of Longbow operators.

patmatch.lisp
  Code for pattern matching used by the unifier code. This code is taken from
  Russel and Norvig [?].

struct.lisp
  The definitions of the principle data structures used by Longbow.

unifier.lisp
  Code for unification. This code is taken from Russel and Norvig [?].

utilslisp
  Code for various plan interface functionality.

variable.lisp
  The code for manipulating variables and variable bindings.

version.lisp
  The code for determining the current version of source code.

E.7  lb:ops Directory

The ops directory is the principle repository for pre-defined Longbow operator definitions.

suite.lisp
  This file contains a number of example domains that indicate some of the plan
structures that Longbow can construct. Domains in this file range from simple ones indicating the use of causal links, constraints, and other basic plan components to more complex examples containing conditional decompositions, constraints and preconditions satisfied programmatically, etc.

**html.lisp**

*Domain Definitions*

This file contains a single domain that implements a straightforward html page generator for describing software systems. The scope of this domain is fairly narrow, but it provides one good example of a domain that constructs a complex plan and then translates that plan into readable output (the output of this domain is readable by most web browsers).

## F Assorted Longbow Data Structures

The following sections detail some of the data structures useful for applications developers interested in Longbow internals. Note that these lists are not complete; some slots in some structures are not documented. Prior to the official Longbow 1.0 release, care should be taken by developers writing custom code to keep direct access to Longbow internal data structures to a minimum. These structures and their access functions are somewhat volatile and may change without notice. Backwards compatibility will be provided where possible.

### F.1 The plan Data Structure

The `plan` structure is used by Longbow to store all information about any plan. Both partial and complete plans are represented using `plan` structures. Plan structures contain the following slots:

- **steps**
  
  *Plan Component*
  
  A list of step ID values. This slot holds the ID numbers of each step in the plan.

- **links**
  
  *Plan Component*
  
  A list of causal links. This slot holds pointers to each causal link in the plan.

- **decomp-info**
  
  *Plan Component*
  
  An association list of information about the decompositions used in a plan.
flaws

Plan Component
This slot holds a list of the plan’s flaws. Each flaw is a structure itself and is either of type OPENC, UNSAFE or P-STEP (corresponding to unexpanded abstract steps). If this slot is nil, then the plan has no flaws (that is, it is completed).

ordering

Plan Component
A list of two-element lists. Each sublist contains two step IDs; a sublist of the form (ID1 ID2) indicates that the step with ID ID1 must occur in the plan before the step with ID ID2.

binds

Plan Component
The positive binding constraints of a plan.

not-binds

Plan Component
The negative binding constraints of a plan.

high-step

Plan Component
The ID number of the most recently created step in this plan. This ID number will be the greatest ID number of any step in the plan.

name

Plan Component
A name slot for internal debugging purposes.

reason

Plan Component
A slot containing information about the refinement that gave rise to the creation of this plan.

F.2 The p-step Data Structure

Each step in a Longbow plan is represented in a p-step structure.

ID

Step Component
A unique integer that is used to identify the step in Longbow code.

action

Step Component
A list describing a formula such as (puton ?X ?Y). The value of this slot is used almost exclusively to generate formatted output describing the step.
act-type

A symbol such as Puton.

parms

Once a p-step is instantiated, variables for that p-step are uniquely identified by having the p-step's ID appended to their names. For instance, if a step’s ID is 5 and the step’s action (see above) is (Puton ?x ?y), then the :parms will be (?x5 ?y5).

precond

A list of conditions like (Clear ?X) representing the preconditions of the step.

add

A list of effects asserted by the step.

effects

The original effects formulae that appear in the operator definition for the p-step’s act-type.

parents

A list of parent step IDs. A step may have more than one parent step when it is used in the subplan of more than one abstract step.

ancestors

A list of ancestor step IDs. The transitive closure of the parents relationship.

children

A list of the step IDs of all steps in the step’s immediate subplan. Note that, since step data structures are shared across plans during the plan construction process, the value of this slot is likely to be inaccurate until plan search is completed. If search using this slot is needed before the complete plan is constructed, the information can be obtained by querying the parents and —it ancestors slots of the steps in the plan.

descendants

A list of all descendant step IDs. The transitive closure of the children relationship. Note that, since step data structures are shared across plans during the plan construction process, the value of this slot is likely to be inaccurate until plan search is completed. If search using this slot is needed before the
complete plan is constructed, the information can be obtained by querying the
parents and —tt ancestors slots of the steps in the plan.

<table>
<thead>
<tr>
<th>Step Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>prim</strong></td>
<td>A boolean flag indicating that the step is either primitive (t) or composite (nil).</td>
</tr>
<tr>
<td><strong>description</strong></td>
<td>A text string description of the action operator.</td>
</tr>
<tr>
<td><strong>template-fn</strong></td>
<td>A function name to be called when generating the text associated with primitive actions (utterances).</td>
</tr>
<tr>
<td><strong>constraints</strong></td>
<td>The action constraints on this step.</td>
</tr>
</tbody>
</table>

**F.3 The effect Data Structure**

Each effect of each step in a Longbow plan is represented by a distinct effect structure.

<table>
<thead>
<tr>
<th>Effect Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>The actual conditions added by the effect.</td>
</tr>
<tr>
<td><strong>forall</strong></td>
<td>A UCPOP vestigial slot. Not used in Longbow.</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>The ID of the step associated with the effect.</td>
</tr>
<tr>
<td><strong>precond</strong></td>
<td>A UCPOP vestigial slot. Not used in Longbow.</td>
</tr>
</tbody>
</table>

**F.4 The dlink Data Structure**

Each decomposition link in a plan is represented in a dlink structure.

<table>
<thead>
<tr>
<th>Decomposition Link Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>children</strong></td>
<td>A list containing the IDs of all the steps in the new subplan.</td>
</tr>
</tbody>
</table>
parent

The ID of the parent step of the decomposition.

schema

The structure of the decomposition operator used to expand the parent step into the subplan containing the children steps.

F.5 The link Data Structure

Each causal link in a plan is represented in a link structure.

condition

The condition associated with the link

id1

The ID of the link’s source step.

id2

The ID of the link’s destination step.

F.6 The unsafe Data Structure

Unsafe causal links are represented using unsafe structures. Like structures that represent other plan flaws (open preconditions and unexpanded abstract steps), unsafe appear in the flaws slot of plans.

link

The ID of the source step of the threatened link.

clobber-effect

The effect that threatens the link.

clobber-condition

The condition threatening the link that is asserted by the clobber effect.
F.7 The openc Data Structure

Open preconditions are represented using openc structures. Like structures that represent other plan flaws (unsafe causal links and unexpanded abstract steps), opens appear in the flaws slot of plans.

```
condition
  The open precondition.

id
  The ID of step with the open condition.
```

Open Condition Component

G Release Notes

- Longbow 1.0 alpha 12

Comments

It has been some time since the previous release. Quite a few low-level improvements have been made to the code, many that should be transparent to the users and one or two that will not be.

As always, a back-up copy of your previous installed source should be saved until you have determined that the newest release does not introduce bugs into your particular application. Also, please be sure to recompile any of your modified source code and test the new system using the recompiled code before reporting bugs in the new version.

1. New listener interface functions have been added to locating and inspecting plans in the space searched by calls to the Longbow planner. These include
   * a function that resumes search through a plan space after the first plan has been located.
   * a number of functions for viewing plans and plan components without having to create arrays of the current plan space.

More information on these functions can be found in section C.

2. The syntax for decomposition operators has been modified (Note: the new version of Longbow is previous syntax, so updating your operator sets to the new syntax is entirely optional).
Previous version of Longbow required that begin and end steps be specified for every decomposition operator, even though the information contained in these specifications was almost entirely redundant. In the new version, begin and end steps are defined implicitly. They are always created for each decomposition and given the names begin and end, respectively. The variable names that had previously been specified in the begin and end step definitions are now specified in a :parameters slot of the decomposition. The values of this slot must match in number and structure the :parameters slot of the action operator that the decomposition decomposes, although the names of the variables may differ.

Examples of the new syntax can be found in the file `lb:ops:suite.lisp`. A complete discussion of writing operators using the new syntax can be found in section 3.3.3. A warning message may be printed when operator sets that use the old syntax are loaded. If you do not chose to update your operators to the new syntax, simply ignore any warning messages. Longbow will continue to support the older, more verbose syntax.

3. In response to a request from Stephan Kerpedjiev, the mechanism used to rank plans in the plan space has been seperated from the mechanism used to determine if a plan is complete. In previous versions, plans were assigned a special ranking of -999 soon as it was determined that they were complete. This special value triggered the search function to return the plan as a solution to the planning problem. Complete plans now retain their ranking values, they may not be the most preferred plans in the plan space, etc. This provides a more expressive representation of the plan space that can be exploited by custom search functions.

4. The syntax for specifying ordering constraints in decomposition operators has been made more flexible. Previous versions required orderings to be expressed pair-wise. As a result, an ordering between three steps \( s_1 < s_2 < s_3 \) required two lists to specify it: \( ((s_1 \ s_2) \ (s_2 \ s_3)) \). In the new version, a total order between \( n \) steps can be expressed using on list containing the \( n \) steps, listed in order corresponding to their position in the temporal order being described. For instance, to describe the ordering \( s_1 < s_2 < s_3 \), one simply writes: \( (s_1 \ s_2 \ s_3) \).
Note that the old syntax is upwards-compatible with the new syntax, so there is no requirement that you modify existing operator sets.

Bug Fixes 1. A handfull of small bug fixes have been repaired.

Persistent Bugs None.

New Bugs None.

• Longbow 1.0 alpha 11

Comments 1. A new dlink data structure has been created to store information about individual decompositions of abstract steps in plans. The information stored in this structure is accessible via the access functions get-decomposition, get-decomposition*, get-children-steps and get-children-steps*, documented in appendix C. Additional access functions will be added as more information is included in the dlink structure.

2. The principle function used to call the Longbow planner has changed from lb-plan to find-plan. find-plan is similar to lb-plan except it takes a keyword argument indicating whether to find one plan in the search space or all plans in the search space. The new function is documented in appendix C and the lb-plan function will remain a defined function (though unsupported) for some time in order to facilitate backwards compatibility. Users should expect lb-plan to be removed before Longbow 1.0 is released, however, and so they should change their code to use find-plan before this point.

3. In addition to the expanded functionality provided by find-plan, users interested in finding more than one plan in a domain one at a time may now use the function continue-finding-plans. continue-finding-plans assumes that plan search has already performed on the current domain and has, in fact, returned at least one successful plan solution. continue-finding-plans resumes plan search from the point that the last plan solution was returned. This function is useful for developers that wish to evaluate the appropriateness of plans after they have been constructed but do not wish to suspend their processing while exhaustive search is performed by find-plan.
4. The function name used to call the CLIM user interface has changed from `lb-clim-ui` to `clim-gui`. The new function is documented in the manual and the old function will remain around, though, unsupported, for some time to facilitate backwards compatibility. Users should expect `lb-clim-ui` to be removed before Longbow 1.0 is released, however, and begin to use `clim-gui` instead before this point.

5. The function name used to call the default plan ranking function has changed from `rank4` to `rank-plan`. The function definition of `rank-plan` is identical to that of `rank4`'s, and the new function is documented in the manual. The function `rank4` is no longer defined by Longbow. `rank-plan` is now the function that users should customize to provide their own plan space search ranking function.

6. Developers should be aware that some internal data structures used by Longbow may change somewhat over time. Access to data via the documented access functions will most likely remain available, but direct access to the structures should be made with care.

7. A new logical pathname has been defined for Longbow users. The distribution tar file will create a directory named `patches` under the top-level `lb:` directory and this logical pathname will point there. For users at a remote location, this directory and pathname may be useful to store patchfiles as they become available for a particular release. For users of Longbow on systems where I install the system source code automatically, the `patches` directory will automatically be updated with patchfiles for you.

Note that in order to take advantage of the new pathname you will need to re-load the file `lb:lb-init.lisp` or copy its contents into your lisp init file.

**Bug Fixes**

1. In previous releases, explicit ordering constraints in decompositions (those appearing in the `:orderings` slots of decomposition operators) were ignored when one or both of the steps in the ordering was a `conditional` step. This bug has been fixed. You can now order conditional steps in arbitrary ways just like any other step in a decomposition.

2. When rewrites in decompositions were intended to rewrite preconditions of end steps of the form `(not /<condition/>)`, Longbow was failing to match the precondition to the rewrite specification and so the precondition was not getting re-written. Often this bug resulted in the
search limit being exceeded during planning as the planner attempted to build an infinite plan. It would decompose a parent action, then add a new action of the same act type to establish the new final step’s mistakenly un-rewritten precondition. This would in turn require the expansion of the newly added step and the cycle would repeat. This bug has been repaired, however, please see the persistent bug list and the new bug list for a description of two bugs that may have similar origins and may not be repaired.

Persistent Bugs
1. Despite our efforts, the status of Mark Derthick’s bug remains unde-termined. We are in the process of trying to recreate the bug, but suspect it has something to do with the failure of rewrites to match on the correct preconditions of the final steps of subplans. Some bugs in this functionality have independently been isolated and repaired, so it may be that this obscure bug has also been fixed. We are currently working with Mark’s operator set to recreate the bug and, should we find it in the new release, we’ll fix it and make a patch available.

New Bugs
1. Some operator sets that use the relation constant :not in preconditions and effects appear to miss some rewrites in decompositions involving those preconditions. Until this bug is repaired I suggest you use some other token in place of the :not constant (for instance, not works fine).

2. For systems using the CLIM GUI package, compiling longbow using the cl-user::compile-longbow function may result in the compiler breaking once during the compile of the clim20.lisp file. Simply continue from the break. This should cause no problem.

3. One user has reported that the load process for Longbow will break several times when loading compiled code. I have not been able to recreate this problem in either MCL or Allegro lisps. I’ll continue to look for the source of the problem. In the mean time, if this occurs to you, simply continuing from the break seems to work. If you do find this problem, please report it to me.

• Longbow 1.0 alpha 10

Comments
1. This version adds a number of new and useful debug printing functions. These are detailed in the manual in Appendix E as well as section 4.4 on debugging Longbow domains.
2. There is now a switch to toggle between developer mode, where useful debugging and trace information is printed during a Longbow run, and production mode, where no such messages are given. See the manual for details.

3. Information has been added to each plan that is created that makes debugging more straightforward. Now users can tell (in many cases) why a plan was created and what should be expected in its structures.

**Bug Fixes**

1. This version fixes a number of bugs in the plan ordering code that was allowing parents of steps to be sources for their children’s causal links.
2. An obscure but deadly bug with variable name conflicts was repaired.

**Persistent Bugs**

1. Mark Derthick’s obscure bug may still persist. This bug will be tested in the new code soon. Expect fixes to appear in the patches directory.
2. Conditional steps mentioned in explicit orderings in decomposition operators are only ordered by the normal ordering constraints – the explicit orderings are ignored and a spurious error message will be printed during the syntax checking of the operator set.
3. In some environments, loading compiled code for the Longbow system will generate several error messages and breaks. Continue from these breaks and ignore the error messages. This will be repaired soon.

**New Bugs**

1. None have yet been reported.

- **Longbow 1.0 alpha 9**

This release contains a number of changes that are considered critical for several alpha sites. Hopefully the new and improved code will justify coming out with a new release so soon.

As far as bug fixes go:

1. Now performs correct syntax checking of the rewrites in decomposition operators.
2. Fixes Longbow’s failure to add multiple plans when several effects of a single step can be the source for a single causal link.
3. Fixes some bugs where plan-orderings were getting corrupted due to the fact that step data structures are re-used across plan siblings.
4. Some compilers broke when reading alpha 8’s file \texttt{lb:code;expand.lisp}. The offending form has been removed.

5. Unfortunately, a few other fixes have slipped into oblivion before I could record them.

There are a few new items to be aware of:

1. A more useful plan inspection function named \texttt{print-plan-tree} has been added. This function prints out a plan via the listener in a hierarchical manner with proper step ordering. Keyword parameters allow the user to specify whether or not the function should also display the open preconditions, and incoming and outgoing causal links of each step. The function accepts a plan which may be partial along with a number of keyword arguments. Reasonable defaults are provided. Time does not permit me to update the manual to document this function further. I feel that the function is fairly straightforward to understand and may be very useful to people trying to determine what plan structures are being generated by Longbow, so I’m including it in this release with minimal documentation. The source code can be found in the file \texttt{lb:code;interface}. Further documentation will appear in the manual as soon as possible.

2. In order to make updating your code easier, I will start using a patch distribution approach. As before, only the most recent release will be supported, but rather than sending out a new release every third day I will instead provide a patches directory in the ftp site for Longbow. The patches directory will contain functions patching various bugs found in the current release. For the latest updated version, load the latest distribution files and then load each of the files found in the patches directory. The function \texttt{lb-version} has been updated to provide information about the current loaded patches. Also, the README file in the patches directory will contain a brief summary of each patch.

The bug that fails to completely order conditional steps in decompositions still persists in this release, as may Mark Derthick’s bug.

- \textbf{Longbow 1.0 alpha 8}
The code is getting much more solid, although Mark Derthick’s old bug may still persist in this release (this release is as yet untested on his domains and his bug doesn’t show up in any of the test domains).

There have been a few bug fixes, including:

1. You are now allowed to have numeric arguments to constraints. Sorry – this was a mistake on my part. The syntax checker was unintentionally flagging numeric args as errors.

2. Variables that are referred to in the rewritten preconditions of subplan finals steps that do not occur as arguments of the final step were not getting unified with or bound correctly when the preconditions that they appeared in were being closed by steps not created in the decomposition. Pretty specific, huh? Well, now that bug is fixed.

3. The code that handles the precondition-kb-query functionality was a) not being called correctly and b) doing silly things when it was called. Apparently not too many people use this feature, but I noticed the bug and have fixed it now.

4. Another bug in causal link creation in decompositions was fixed.

5. A left-over break function that printed the helpful message “oops” has been removed.

6. Some bugs that kept some explicit ordering constraints in decompositions (those ordering constraints that are listed explicitly in the :orderings slot of the operators) from being added to the plan have been repaired.

7. Additions to the syntax checking code include checking the :orderings constraints in decompositions.

8. Syntax checking for longbow operators continues to improve, but remains incomplete. The next release will contain even more checks to make sure that errors in operators do not give rise to odd planner behavior.

The other major change was the expansion of the sample domains distributed with the release. Now all example domains are found in a file named lb:ops:suite.lisp, including the parts domain from previous releases and several domains that are based on that one but produce more complicated plans.

The plans produced by the domains appear to me to be correct and I now use them and a few other plans as test-cases for each new release. These domains
are meant to be representative of some but not all of the plan structures that Longbow can produce. Feel free to play with the examples, modify them and see how and why Longbow deals with your modifications.

Along with any remaining bug fixes, it is a high priority to produce some means of inspecting plans and their components. This will mean fixing up the CLIM 2.0 interface as well as writing some additional code to browse plan structures via the listener.

There is one known bug in this release. Orderings in a decomposition that arise only by virtue of their appearance in the ordering slot are ignored if either of the steps mentioned in the ordering constraint is a conditional step. This does not mean that those conditional steps will have no ordering constraints placed on them – constraints arising from causal links or from restrictions that the steps appear between the bounding steps of the subplan will be added to the plan as usual.

This bug will be fixed in the next release.

A small modification has been made to the function text-display-plan. This fn now displays each step’s parent step when displaying a plan’s components.

- **Longbow 1.0 alpha 7**

  There is now only one remaining known bug, and it’s possible that this bug has been fixed by this release. Until new bugs are reported, development will focus on the following:

  - Creating a test suite of domains for a) verifying Longbow functionality and b) demonstrating its plan generation features.
  - Documenting lisp functions that are useful for inspecting the plans produced by Longbow.
  - Improving the CLIM interface.

Bug fixes in this release include:

1. What I suspect were the last of the bugs that were causing problems with variable binding in decompositions have been removed.
2. What I hope were the last of the bugs that were causing errors in conditional causal link creation in decompositions have been removed.
3. Additional syntax checking has been added to Longbow. There has NOT been a change to the operator syntax, but now some errors not previously caught by the syntax checker will be flagged.

4. Some corrections to the manual have been made. In particular, the specifications for the kb-query functions now reflect the reality that those of you who have been using them have known all along. Sorry for the misleading documentation and thanks to Barbara Di Eugenio for pointing this out.

- **Longbow 1.0 alpha 6**

1. There is now a small change in the action operator syntax. In the new syntax, instead of defining the operator as a “operator” you must define it as an “action.” That is, instead of

   ```
   <action-operator> ::= (define (OPERATOR <operator name>)
                        <the rest of the definition>)
   ```

   you now use

   ```
   <action-operator> ::= (define (ACTION <operator name>)
                        <the rest of the definition>)
   ```

   In the old syntax, action operators were defined as operators, while decomposition operators were defined as decompositions. The new syntax is more symmetrical with respect to the names of the two types of operator definitions; hopefully this will be less confusing to future users. See section B for the exact specification. Thanks to Pam Jordan for this suggestion.

2. Ah, the bug fixes:

   (a) There was a bug in the functions that selected operators useful for establishing open preconditions. The matching algorithm was overly generous and as a result, some plans may not have been unsound. This has been fixed.

   (b) Some causal links from conditional steps to the end step of a decomposition were not being added to subplans. This would cause the planner to a) halt without having added the link or b) to create extra steps to close the conditions that the missing link was supposed to have closed. This bug has been fixed.
(c) Some bindings resulting from the addition of causal links in decompositions were not being added. As a result, some plans were being returned that had unbound variables for some steps in some subplans. This bug has been fixed.

(d) Some preconditions of some final steps in subplans were flagged as open when actually they were oeclosed. This resulted in plans being returned had too many steps or unneeded causal links in some subplans. This has been fixed.

(e) Some preconditions of some final steps were being flagged as closed when in fact they were open. Specifically, preconditions of final steps that were arose from rewrites were not being marked as open conditions. As a result, some plans were being returned as completed when in fact they had open preconditions of some final steps in some subplans. This bug has been fixed.

(f) Some steps added during action decomposition were not having their action constraints checked when they were being added. This has been fixed.

3. Known bugs:

(a) Some conditional steps in action decompositions are not having their action constraints checked. This will be fixed in the next release.

(b) Pam Jordan's bindings bugs may have been repaired by the latest revisions. I will do further testing to verify this.

(c) Mark Derthick's infinite decomposition bug may also have been repaired. I'll try to verify this as well.

- Longbow 1.0 alpha 5

Well, as most alpha test sites noticed, the source code files used to create the tar file for alpha 4 were incomplete, resulting in an unexecutable system. Alpha 5 fixes this problem as well as a number of by-now familiar bugs. These include:

1. The most persistent of bugs, the one that caused redundant steps and causal links to be added to some decompositions when the preconditions of the decomposition's final step were incorrectly flagged as open (even though they had been explicitly closed), has now been fixed.
2. A new bug that incorrectly named internal variables in different decompositions in a plan by the same names has been fixed. This bug caused problems whenever two or more decompositions of the same act type were used in the same plan, since the bindings associated with the temporary variables used in the first decomposition would interfere with the satisfaction of the constraints in the second decomposition.

Some know bugs still persist. These include:

1. Some variables in decompositions are still not being assigned bindings.
2. In rare cases the planner appears to decompose actions over and over again in an incorrect manner, making the planner exceed its search limit and building an incorrect plan space in the process.

Finally, a number of improvements are expected for the next alpha release. These include:

- The fixes to the known bugs mentioned above.
- Updated documentation describing the recent changes in decomposition operator syntax.
- Documentation describing lisp functions useful for inspecting plans and the plan space. These functions are of particular use to developers passing parts of the plan structure to subsequent components in a generation or execution process.
- Some minor repair to the CLIM2.0 interface.

• **Longbow 1.0 alpha 4**

A number of old and new bugs were fixed in this release. A small number of new bugs were found but have not yet been fixed. Also, some restrictions on operator syntax have been added. See the discussion below for details.

**Bug Fixes** (a) In previous releases, the satisfaction of decomposition constraints from the initial state (that is, when *longbow-constraint-kb-flag* is nil) did not work correctly. This bug has been fixed in this release.
(b) In previous releases, the satisfaction of action constraints (either via the initial state or via the functions `constraint-kb-query` and `constraint-kb-query*`) did not work correctly. These bugs have been fixed.

(c) A bug that caused the system to break when a variable in a decomposition conditional was bound to `nil` has been repaired.

(d) A bug that caused the system to break in some situations where a decomposition’s constraints failed has been repaired.

(e) A bug that caused the system to break when some decomposition operators specified more than one conditional step definition has been repaired.

(f) A number of other small bugs have been repaired.

Persistent Bugs

(a) The bug that causes additional steps to be added in some decompositions (in order to close final step preconditions that are explicitly closed in the decomposition operator) remains.

(b) Some variable bindings are not added in some (increasingly rare) cases.

New Open Bugs

(a) Apparently there is a bug that causes some planning problems with conjunctive goals in the `:goals` slot of the planning problem to fail to find solutions to the problem when they exist.

(b) Some action constraints on steps in decompositions are not being checked when the expansion is performed.

Syntactic Restrictions

(a) Variables in decompositions that are bound by the constraint-satisfaction process to lists must only appear a) in the `:constraints` slot of the decomposition or b) as the list-bound variables in conditionals. That is, these variables may only be used as the variable named `<list variable name>` in the form below:

    (forall <variable name> in <list variable name>
        (form-1)
        ...
        (form-n))

These list-bound variables may not be appear as arguments to steps in the decomposition or be bound via decomposition binding constraints to code designate with step arguments.

- Longbow 1.0 alpha 2
1. The mechanisms for checking a knowledge base to satisfy preconditions from the initial state and to satisfy constraints of any kind have been separated out. This allows the user to specify different functions for precondition and constraint satisfaction. See section refsec:kb-query for an explanation of the differences and how to deal with them.

2. The functions new-kb-query and new-kb-query* have been removed and replaced with the functions constraint-kb-query and constraint-kb-query*, respectively. See section 5 for specifics.

3. A number of bugs that resulted in bindings not being established for steps in decompositions have been removed.

4. A bug that caused Longbow to break when an operator that defined more than one conditional step was used has been repaired.

5. In previous versions, some variables in conditional steps there were not the variables used in the conditional expansion were not being bound properly. This bug has been removed.

6. The alpha 2 version contained a bug that caused Longbow to break when dealing with conditional steps of any kind. This bug is fixed in this release. The alpha 2 version contained a number of annoying format statements used for debugging that have been removed in alpha 3.

7. At least one serious known bug remains in this version. Some preconditions of some subplan final steps are not recognized as closed even though the decomposition explicitly adds links to close them. This will be repaired real soon now.

8. Code to indicate the current version number of Longbow has been added. Code has also been added to view any incidental notes related to the current version. See the functions lb-version and lb-version-notes in section C.

- Longbow 1.0 alpha 2

1. The order in which variables are bound during subplan creation has been changed. Now the variables used in the final step specification in a decomposition are bound to the corresponding variables in the parent step before the constraints are checked, so that the constraint checking code has access to these bindings.
2. Some of the source code has been re-written to be more modular, efficient and readable.

3. A number of errors in the manual have been fixed, including mistakes in the EBNF forms for decompositions. The actual operator syntax remains unchanged.

4. A serious bug which prevented bindings from being added when some new causal links were drawn has been fixed. A number of smaller, less obvious bindings bugs were also repaired.

5. A number of known bugs remain and are expected to be repaired in the short term (days). These bugs include further variable bindings problems, the problem of the addition of steps and links that redundantly close preconditions of subplan final steps and some problems associated with decomposition operators that specify more than one conditional rewrite.

- Longbow 1.0 alpha 1

1. This code fixes a number of bugs. First, conjunction in the action and decomposition operators can now be specified as defined in the EBNF forms, rather than using the older \texttt{and} forms work-around from 1.0a1. Second, a number (but not all) of the bugs that caused binding constraints to be left unspecified have been fixed. A number of these remain, however, and are being tracked down currently.

2. In addition to the remaining bindings bugs, several known bugs remain and are being currently being fixed. First, some preconditions of final steps of subplans that are created by conditional rewrites may be considered open even after they have been closed by conditional links. As a result, some extra steps may be added to some subplans (in order to close these conditions).

- Longbow 1.0 alpha 1

1. Finally, this code is, as the name indicates, the \textit{first alpha release}. The code is certainly not pretty. It is likely not to be bug-free. It probably is inefficient in parts. \textit{Any} comments or questions will be welcome at longbow@isp.pitt.edu. In fact, your comments are essential to the improvement of the Longbow code and the successful development of subsequent versions.
2. Currently, the alpha distribution contains a large amount of code left over from UCPOP 2.0, the non-hierarchical planner on which Longbow is based. Much of this code is not used and only serves to confuse those of you interested in the details of the Longbow implementation. In addition, code supporting previous (unreleased) incarnations of Longbow remains in this distribution. All dead code and all data structures not used by Longbow will be removed in the next release.

3. The Macintosh graphical user interface is not meant to provide the same look and feel of the CLIM 2.0 GUI. The Macintosh interface takes a minimalist approach to interface design, relying heavily on the Macintosh Common Lisp 2.0 graphical inspector's ability to display data structures. Future releases may or may not expand the Macintosh user interface.

4. The CLIM 2.0 interface is currently in a state of flux. This code is still in the process of being ported from CLIM 1.1, so much of the look and feel retains CLIM 1.1's limitations. This should change by the next release.

5. Negation in preconditions, effects and constraints is currently dealt with like any other construct in the condition language. Unification is used to match any two conditions and to determine variable bindings, and the relation symbol \texttt{not} is not treated any differently from any other. As a result, negated conditions can effectively be used, but not all threats to causal links will be detected. In future releases, negation will be returned to its special status, allowing a full range of threat detection.

\section*{H Proposed Future Enhancements}

Note: this section is highly speculative. Absolutely no commitment towards implementing any of these modifications is implied. They are listed here for the reference of the Longbow development team and those developers from the the alpha test sites who have contributed these ideas.

\subsection*{H.1 Unification}

1. Threats and a customizable threat detector and threat resolver.

2. Modification of unifier to deal with negation and non-codeignation constraints.
3. Customization of the unifer for different applications.

H.2 Operator Syntax

1. Implicit start and end steps and a :parameters slot on decomposition operators.

2. Arbitrary condition expressions as parameters to steps in :steps specifications in decomposition operators.

3. Multiple conditional components with forall constructs in decomposition operators.

References


