Lisp Object State Saver (LOSS)

A Facility Used to Save Partial Schedules of the Hubble Space Telescope

Jeffrey L. Sponsler
Space Telescope Science Institute
3700 San Martin Dr.
Baltimore, MD 21218

Abstract

Current research in the area of long term scheduling of the Hubble Space Telescope is being done using Common Lisp and Flavors on Lisp Machines. The planning tools manipulate memory-resident data structures which represent the many entities and relationships that represent planning states. The Lisp Object State Saver (LOSS), a general purpose utility, has been constructed which allows one to take a snapshot of memory by storing a representation of the structures in a text file. This text file can later be loaded thus restoring the pre-existing and logically equivalent planning state. A LOSS template must be created for each datatype to be stored and a simple grammar governs the creation of such templates.

1Operated by the Association of Universities for Research in Astronomy for the National Aeronautics and Space Administration
Contents

1 Introduction 3
   1.1 What is it? ................................................................. 3
   1.2 Why would one want to use such a facility? ....................... 3
   1.3 LOSS is a tool used to enhance SPIKE .............................. 4
   1.4 How is this state saver used? ....................................... 4

2 Using LOSS to save a Lisp State 5
   2.1 The State is the focus of processing, ............................... 5
   2.2 The State Manager is the interface to LOSS ....................... 5
   2.3 How does LOSS work? .................................................. 5
   2.4 Templates are the key to state save/restore actions. .......... 7
   2.5 Saving a state ........................................................... 8
   2.6 Form of a State File .................................................... 9

3 Restoring a state file to memory 9

4 Intended improvements 10

5 Acknowledgements 10
1 Introduction

1.1 What is it?

The Lisp Object State Saver program was designed as a mechanism that allows one to store memory-resident data structures in a LISP environment to an ASCII file. This file can later be loaded into memory which would be restored to a logically equivalent pre-save state.

At first glance this is not much of a problem for, say, a simple variable \( x \) that is bound to an integer. However, certain data structures often have references to other more complex data structures (eg., LISP structures and flavors). Such a reference is a pointer (internal address) and therefore cannot easily be stored as character data that would be meaningful upon memory restoration.

1.2 Why would one want to use such a facility?

In a LISP environment, it is often the case that one creates large numbers of data structures that are organized into networks with rich and complex interconnections. Such networks have been used to represent knowledge about a wide variety of domains (e.g., medicine, factory scheduling, etc.). The data structures themselves represent the nodes (entities) and their attributes store pointers to other data structures and thus represent the arcs (relationships) in a directed graph model. Such memory resident knowledge bases are the rule in artificial intelligence applications.

The creation of semantic networks generally is accomplished by loading a file that contains LISP forms that upon evaluation instantiate the data structures. Programs can then be executed that (as side effects) will create logical interconnections that dynamically form the network(s).

Often this can take a considerable amount of real time. In many LISP environments, once such a large and richly connected memory state is formed, it is advantageous to save the entire program memory state to a binary file on a hard disk medium (as opposed to recreating all structures and connections again). This file later can be reinstated as the memory of the LISP interpreter.\(^2\) Although this works well there are disadvantages.

1. If one does not effect garbage collection before the save, a large amount of disk space will be used to save uncollected garbage (memory locations that are no longer referenced).

2. These binary files are often huge and it is not possible to store very many of them.

3. It may be impossible to take the binary image file from one lisp environment (e.g., an Explorer) and load it into another (a SUN).

4. This mechanism stores not only the relevant knowledge base but also the Lisp interpreter, and data structures that are associated with the operating environment but irrelevant to the precise application that is the focus of one’s work.

\(^2\)On a LISP machine, the file is a bootable braid; on a conventional architecture, the file is an executable image.
With the above problems in mind, this system has been designed and implemented.

1.3 LOSS is a tool used to enhance SPIKE

The SPIKE system is being developed at the Space Telescope Science Institute to support the creation of long term schedules for the Hubble Space Telescope [1].

Briefly, the SPIKE system prototype currently resides on a Texas Instruments Explorer computer and is implemented in Common Lisp and MIT Flavors. A mouse/menu/graphics-oriented interface is provided for the user.

The central concept of the planner is the suitability which is implemented as a piecewise constant function; such a structure is a list of time points and values that represent how well an activity would schedule over time. Entities represented as flavors in the system include targets (stars, planets), activities (exposures, acquisitions, calibrations), segments of time, and constraints (sequential offsets, separation, sun exclusion, etc.). A group of activities may be grouped into a scheduling cluster, which is then considered a unit (the suitabilities of all activities within contribute to the cluster suitability).

By various means (manual or automatic selection) one may commit a cluster to a time segment. The act of commitment may often change the suitabilities of other scheduling clusters.

After a planner (a human who is working on an HST schedule) has worked with the SPIKE system for some period of time and possesses a partial schedule of some merit, the LOSS system can be invoked to store a snapshot of memory to a file. The planner may then continue working with the schedule that remains in memory (and may file other states). If at some point, a scheduling session has reached a point that is undesirable, there are two options. The first entails using the SPIKE planner to undo commitments in an order which is the reverse of the order in which they were made. The second option can be invoked if the planner wishes to return to a much earlier state; SPIKE operating memory is first cleared and then the LOSS State Manager is employed to select a state that had earlier been saved to file and to restore that state to memory.

1.4 How is this state saver used?

The state saver is able to save most types of data structures as logical character representations in a specified file. Currently it is possible to save the following: global variables, atoms, numbers, lists, strings, hash tables, structures, and flavor instances. The mechanisms to save arrays, property lists, and association lists are not yet in place.

It also has the ability to restore those data structures to a state that is logically equivalent (based on pointer interconnections) to the pre-save state. Only the data structures and attributes of such structures that the user specifies are saved. The file character representation that is the result of a save is generic and machine-independent; this system does however operate currently only in a Common Lisp environment with a Flavors system resident.
In order to facilitate the organization, filing, and restoration of states, a State Manager has been designed and implemented using Lisp, Flavors, and Common Windows \(^3\).

These steps involved in using LOSS are detailed with examples in the following sections.

2 Using LOSS to save a Lisp State

Following is a brief discussion of the main elements of the operating LOSS environment. A description of how the state saver works and a simple example are included.

2.1 The State is the focus of processing.

In order to effectively manage the saving and restoring of Lisp memory states, the concept of the state has been implemented as a data structure. A state tree can be created that organizes the states into a strict hierarchy. The attributes that can be associated with each state node include the following:

1. Parent and child links for tree traversal.
2. A comment that serves to describe the state.
3. A state file string that names the disk file where the state resides or will reside.
4. A template file where the appropriate LOSS templates are stored.
5. A data-list of structures to save.

2.2 The State Manager is the interface to LOSS

The State Manager is an interactive user interface that has been developed to facilitate the processing of states. It employs menus and windows and allows one to traverse the state tree, to select state nodes for processing, to create and delete nodes, and to initiate a state save or a state restore.

2.3 How does LOSS work?

To illustrate the use of this system, consider a simple example database that describes tropical fish in an aquarium. Following is the code that would define the data structures:

\(^3\)Common Windows is a package that is available from Intelllicorp.
;; Aquarium example data definitions:
(defvar *AQ* nil)
(deffstruct fish name genus species aquarium)
(deffstruct aquarium
  (volume 0)
  (fish-list nil)
  (shape nil))
(defun create-fish (name genus species)
  `(setf (aquarium-fish-list *AQ*)
    (cons (make-fish :name ',name
      :genus ',genus
      :species ',species)
      (aquarium-fish-list *AQ*))))

The following code will create an aquarium data structure and some fish structures.

;; Instantiate data objects into memory -
(setq *AQ* (make-aquarium :volume 30 :shape 'rectangle))
(create-fish clown-loach botia macracanthus)
(create-fish neon-tetra hyphessobrycon innesi)

When this code has been executed, the memory resident data structures will include the following:

#<aquarium 41340377> is a aquarium
  VOLUME: 30
  FISH-LIST: (#<fish 41340432> #<fish 41340421> #<fish 41340410>)
  SHAPE: rectangle

#<fish 41340432> is a fish
  NAME: neon-tetra
  GENUS: hyphessobrycon
  SPECIES: innesi

The data structures for #<fish 41340421> and #<fish 41340410>) will also exist. (The pound-sign notation seen in these structures is a lisp convention that represents a non-printable pointer reference.)
2.4 Templates are the key to state save/restore actions.

LOSS depends upon templates in order to correctly guide the save/restore processes. A template is itself a data structure that contains precise information about data structures to be saved or restored. For advanced datatypes defined as structures or flavors each slot\(^4\) that contains a pointer to a complex data structure is described as well. Simpler data structures such as strings, atoms, and lists do not require explicit save/restore specifications. At the present, templates must be created manually by a software engineer who is familiar with the logical interconnections of the application data structures. The lisp forms that create the templates needed to save the aquarium database follow.

```
(create-template aquarium
  :datatype 'structure
  :no-save-slots '(shape)
  :substitution-slots '((fish-list (list-of (structure fish)))))

(create-template fish
  :datatype 'structure
  :substitution-slots '((aquarium (structure aquarium))))

(setq *IMPORTANT-SPECIAL-VARIABLES* '((*AQ* (structure aquarium))))
```

The templates created above are used to save and restore instances of the structures aquarium and fish. Before such an instance is saved, this template is used to make appropriate decisions about what to save and how. During a restore, the template is used to guide the recreation of the pre-save data structure. Generally, templates are used to specify the datatypes of values that can be placed in slots.

The general syntax of template creation forms is:

```
(create-template <datatype name> <keys-init>)
```

The `<datatype name>` must be the identical symbolic name of the same structure or flavor defined in the application.

The `<keys-init>` portion of the form may contain various specifications for the template concerning the datatype. The :no-save-slots specification must be a list of symbolic slot names whose values the user does not want to be saved to file. In our example, the slot shape is not to be saved. The :substitution-slots specification should be a list of sublists where each sublist has the form (<slot> <substitution spec>). The <substitution spec> adheres to the following grammar:

\(^4\)The term slot will be used to represent the attributes of structures and flavor instance variables.
<substitution-spec> ::=  
  <structure-spec> | <flavor-spec> | (list-of <substitution-spec>) |  
  <hash-table-spec> | (dotted-pair <substitution-spec> <substitution-spec>) |  
  <substitution-function-spec>

<structure-spec> ::=  
  structure | (structure <struct-name>)
<flavor-spec> ::=  
  flavor | (flavor <flavor name>)

<hash-table-spec> ::=  
  hash-table | (hash-table [key-spec] [val-spec])
<key-spec> ::=  
  (key <substitution-spec>)
<val-spec> ::=  
  (value <substitution-spec>)

<substitution-function-spec> ::=  
  (substitution-fn [(save 'sav-fn)] [(restore 'rest-fn)])

It may be the case that a user may wish to process the contents of a slot before a save (or  
after a restore). LOSS templates provide for this in the form of the substitution-fn. The  
substitution function is defined by the template-designer and should be specific to a problem  
slot. This function is called with the contents of the slot and returns the precise lisp form that  
the template designer wishes to have stored. Such functions should be the exception because  
the substitution grammar is general and should be sufficient for most data structures.

It is possible to create specifications that guide the processing of global variables during the  
save and restore activities. In our example the global *AQ* is noted as being bound in the  
application environment to a structure of type aquarium.

2.5 Saving a state

To save a memory state to file, using the State Manager interface, one selects a desired state  
node that references the appropriate template file and state file. Selecting the appropriate menu  
item will initiate the saving. The State Manager will then, using the templates, store the data  
structures to file.

The selected state node should contain a list of pointers to data structures that should be saved.  
Each of these is processed in turn. Consider the variable *AQ* that points to the aquarium  
instance. The pointer cannot be saved to file so it is paired with a unique symbolic index that  
takes the form index25. Both the pointer and its index are placed into a hash-table with the  
pointer as key. In this way, if any other data structure references that pointer, a substitution  
can be made such that index25 replaces the pointer. Since all indices are simple identifiers, one  
can store them as character data in a file.

The type of the structure is found to be aquarium and it is determined that there is a tem- 
plate for this structure. The aquarium structure has slots volume, fish-list, and shape; the
contents of these slots must be considered. The aquarium template is accessed and it is found that the slot \texttt{shape} should not be saved.

Each slot that has a substitution specification in the template is processed using that specification. Generally, pointer reference components of the contents of such a slot are replaced with symbolic indices. The contents of a slot that does not have a template substitution specification are saved to file unchanged. In our example, the \texttt{fish-list} slot has a specification that requires substitution of the slot contents. Each \texttt{fish} instance found there is replaced by a unique index.

\section*{2.6 Form of a State File}

The lisp forms in a state file are generic, machine-independent, and reflect the logical links between the data items using unique index symbols. In the aquarium example, the state file would contain these forms:

\begin{verbatim}
(create-datum index25 structure AQUARIUM
   :volume 30
   :fish-list '(index26 index27 index28))
(create-datum index28 structure FISH
   :name 'leopard-cat
   :genus 'corydorus
   :species 'julii)
(create-datum index27 structure FISH
   :name 'clown-loach
   :genus 'botia
   :species 'macracanthus)
(create-datum index26 structure FISH
   :name 'neon-tetra
   :genus 'hyphessobrycon
   :species 'innesi)
(setq *A0* 'index25)
\end{verbatim}

\section*{3 Restoring a state file to memory}

To restore data structures from a state file to memory, one uses the State Manager to select the desired state node. This node should be associated with the desired state file and appropriate template file. A menu selection will initiate the action.

The state file is loaded and data structures are created. As the data structures are recreated from the state file a new hash-table is built; for each structure, the index will be assigned to the table as a key and the pointer to the structure will be assigned as the corresponding value.
Next, the templates are consulted to determine which slots are to be processed. Any index that is found that is part of the slot’s contents will be replaced with the actual pointer (that is obtained from the hash table). The result should be a restored memory state.

4 Intended improvements

Currently the major weakness of the LOSS system is that the templates must be generated manually. Future versions will include automatic or semi-automatic template generation. Arrays, property lists, and association lists will be represented as saveable datatypes.

The developers also intend to port the LOSS prototype to the Common Lisp Object System when that standard has been set. Since CLOS supports explicit typing of slots this should facilitate the automatic generation of templates. The port to CLOS will also make LOSS more universally available.

5 Acknowledgements

The author wishes to thank the following persons for their comments and ideas related to the work described in this paper: Mark Johnston, Glenn Miller, and Shon Vick.

References