S7-61

N96- 12940

P. 9

An Object Oriented Extension to CLIPS

Clifford Sobkowicz

Government of Canada, Dept. of the Environment
McGill University, School of Computer Science

April 25, 1990

Abstract

A presentation of a software sub-system developed to augment CLIPS with facilities for object oriented knowledge representation. Functions are provided to define classes, instantiate objects, access attributes, and assert object related facts.

This extension is implemented via the CLIPS user function interface and does not require modification of any CLIPS code. It does rely on internal CLIPS functions for memory management and symbol representation.

1 Introduction.

CLIPS (C Language Integrated Production System) is an expert system shell which represents knowledge by production rules which can be applied to asserted facts. Rules represent constant knowledge of relationships between antecedents and consequents, such as causes and effects. Facts specify current information and are either asserted initially, interactively, or as the consequents of rules.

Objects are abstractions of knowledge about hypothetical entities. They are represented as sets of attributes, which can take numeric or symbolic values, and methods for for manipulating them. Objects are members, or instances, of classes with common sets of attributes and methods. Each instance has specific values for the attributes associated with its class. As attributes and methods are qualified by specific objects polymorphism is provided for, whereby actions upon objects can be affected by different means depending on the class of the object.

The capabilities presented here provide for extending rule consequents to include object manipulation and allow for antecedents based on objects and their attributes as well as asserted facts. Also, assertion of facts about objects is facilitated.

2 CLIPS rules and facts.

Rules in CLIPS are composed of a set of antecedents termed the left hand side (LHS) and a set of consequents termed the right hand side (RHS). Facts are ordered sets of fields which can assume single word, numeric, or quoted character string values. The antecedents of rules are patterns which are matched against the current set of facts. They may include wildcard fields, variables which are bound to one or more field values from matching facts, and logical expressions for constraining field values. The consequents are actions such as asserting subsequent facts or side effects such as outputting messages and variable values.

An important feature of CLIPS is the facility for invoking external functions on either the LHS or the RHS of a rule. On the LHS functions can provide data for pattern expansion or be implemented as predicate functions to constrain fields or test conditions. On the RHS functions can perform side effects as consequents of rules. It is via these facilities that objects are created, manipulated and accessed.

3 CLIPS Objects.

Objects in this sub-system are sets of named attributes which can take word, numeric or string values. They can also take multiple values. Attributes are specified by an object name and an attribute name. The data type of an attribute is set dynamically.

Objects include methods which can manipulate the attributes. These are C functions which are integrated with CLIPS, like other external functions, via the `usrfuncs` routine. Methods are invoked by specifying the name of an object and a method selector in an invoke command, along with any parameters to be passed to the function. Also, functions can be attached to attributes and invoked automatically when the value is set or read. Different functions can be invoked from different objects by the same attribute name.

3.1 Implementation.

Functions such as creating objects and accessing their attributes, are implemented by external functions called from CLIPS rules.

Objects are designated by names which are CLIPS symbols and reside in the CLIPS symbol table. They are identified by hash pointers so string manipulation is averted. Objects are implemented as structures in a second table structured after the symbol table. The randomized bucket number from the symbol table entry is used in the object table so as to speed searching. The location of the last object referenced and the last object modified are retained, so performance can be optimized by grouping commands which reference the same object. The object structure includes the hash pointer of the name, a pointer to a list of attributes and a pointer to an inheritance list.

The attributes are stored in a linked list of structures which indicate the type of the attribute: class, instance, or method, the type of the current data: word, number, string, or multiple, the data itself: a pointer or floating point value, the attached functions: two pointers into the CLIPS function table, and some information related to inheritance.

Multiple field data is stored as a linked list referenced from an attribute value element.

4 Classes.

Classes specify sets of attributes and methods common to groups of objects referred to as instances of the class. They are represented by objects that are used as templates for instantiation. Attributes are either class or instance attributes. Class attribute values are maintained in the class object and are common to all instances of the class.

5 Inheritance.

Classes can inherit the attributes and methods of other classes. Thus general super classes can share their functionality with more specific sub-classes which can add additional functionality. Multiple inheritance is provided for in that a class can inherit from a number of classes allowing functionality from general utility objects to be mixed in with super class and local resources. Inherited classes can include inherited resources themselves to unlimited depth. Circular inheritance is disallowed.

A priority can be specified for each inheritance to resolve conflicts when the same name appears in more than one contributing class. The inheriting class carries priority 100 so that inheritances with priority less than or equal to 100 preserve the original resources while those with priority greater than 100 can replace them.

As an alternative to conflict resolution, methods can be declared as multiple in a class definitions. An object can then inherit a list of procedures under a single method name. All of the procedures will be called in sequence when the method name is invoked. It is the responsibility of these procedures to limit their results to non-conflicting side effects such as asserting facts or updating separate attributes or to implement a combining algorithm such as summing or appending results.

5.1 Implementation.

Each class includes an ordered list of inherited classes. The order is that in which the inheritances where specified. The list entries reference the inherited object and indicate the priority of the inheritance. Inherited class objects can inherit classes so the composition of a class is a tree structured set of class objects.

6 Instances.

Instances of classes represent specific objects by maintaining specific data in instance attributes. They inherit all attributes and methods of the class of which they are an instance.

6.1 Implementation.

When a class object is instantiated a new instance object is created. All of the attributes of the class object are copied to the new instance, and assigned a priority of 100. The tree of class objects inherited by the specified class is traversed depth first. For each attribute encountered, if the attribute is not yet present in the new instance it is copied and assigned the priority of the inheritance. If the attribute is already present but the priority of the inheritance is higher than the priority recorded in the instance it is overwritten.

In the event of equal priority, precedence is given to inherited objects according to the order in which the inheritance was specified and class objects are considered to include their inherited attributes.

When instance attributes are copied to the instance object the value in the class object is copied along with pointers to any `when-read` and `when-set` functions. Thus the attribute in the class object serves to provide an initial value and attached functions.

When class attributes are encountered a pointer is placed in the instance object which refers back to the class. Thus the data and attached functions remain common to all instances of the class.

A Appendix: CLIPS commands.

The following are illustrations of CLIPS statements which create and manipulate objects. Familiarity with CLIPS as documented in [1] is assumed.

A.1 Overview.

CLIPS interfaces with the object oriented programming extension by LHS functions and RHS commands implemented as external functions. The function arguments *object*, *class*, *instance*, *attribute*, *method*, or *function* refer to names which must be of type word. Attribute values can be of any type. Parameters for methods or attached procedures are subject to the protocols of the user supplied function.

Methods and attached procedures must be declared as external functions in `usrfuncs`. The *function* parameter refers to the CLIPS name declared for the function.

A.2 Object manipulation.

Classes are defined by a `defclass` construct which specifies attributes, methods, attached functions, and inherited classes.

Instance objects are created by instantiation of a class. They inherit all attributes, methods, and attached functions of the class.

Classes

```
(defclass class "comment "

(methods
(method function) (method function multiple) ...)

(instance-attributes
attribute[<-value] (attribute[<-value] [(when-set function)][(when-read function)]) ...)

(class-attributes
attribute[<-value] (attribute[<-value] [(when-set function)][(when-read function)]) ...)

(inherits
(class priority) class   ...)

)
```

Creates a new class with the given name and the specified methods and attributes. The given function names must be the CLIPS reference names as specified in `usrfncs`. Multi-field values are enclosed in parenthesis. Inherited classes must be already defined.

Instances

```
(instantiate class instance [attribute<-value]... )
```

Creates a new instance object for the specified class giving it the specified name. Optionally specified attributes are initialized. The name of the new instance is returned as the function value so that `gen-sym` can be used to create instance names which can be bound to variables as the function value. The new instance becomes the current object and current instance.

(`delete-instance` *instance*)

Removes instance objects from the symbol table and releases their memory.

A.3 Attribute manipulation.

Valued attributes can be updated and referenced. Methods can be invoked.

Setting values

(`set-attribute` *object attribute [datum [parameter...]]*)

Sets the value of the specified attribute, or flags it empty if no data given. Any previous data is deleted.

The specified parameters are passed to any `when-set` procedures.

(`append-to-attribute` *object attribute datum*)

Appends the given datum to a multi-field value. If the attribute was not previously a multi-field value its contents, if any, becomes the first field. Appending does not invoke any attached procedures as it is not known if the multi-field value is complete.

Retrieving values

(`get-attribute` *object attribute [parameter ...]*)

Obtains the value of the specified attribute. Parameters if given are passed to any `when-read` attached procedures.

Invoking methods.

(`invoke` *object method [parameter...]*)

Invokes the function for the selected method of the specified object and passes it the given parameters. If the method was found in more than one inherited class only the first inherited with the highest priority is called unless the method was specified as `multiple` in which case all multiple designated functions are called.

A.4 Predicate functions.

Predicates about objects can be used to constrain patterns or as tests in the LHS of rules. They can thus prevent rules from making erroneous assumptions about objects.

Attribute of object?

> `(test is-attribute` *object attribute*`)`
>
> Determines if the named attribute is in fact an attribute of the specified object.
>
> `(test is-attribute-numberp` *object attribute*`)`
> `(test is-attribute-wordp` *object attribute*`)`
> `(test is-attribute-stringp` *object attribute*`)`
>
> Determines if the attribute is of a specified type. Returns *false* the specified object does not have the specified attribute.

Inheritance?

> `(test inherits` *class object*`)`
>
> Determines if the named object inherits the attributes of the specified class. The inheritance tree above the object is searched for the class.

A.5 Fact assertion

Facts about attribute values can be asserted into the CLIPS fact list so as to relate knowledge represented by objects to the inference engine. The facts are of the form `(`*object attribute value attribute value* ...`)`.

Single assertion

> `(assert-attribute` *object attribute* ...`)`
>
> Asserts a fact giving the object name followed by ordered pairs of attribute name, and attribute value. For a multi-field value all fields are reported following the attributes name.
>
> `(assert-instance` *object*`)`
>
> As above for all attributes of an instance.

Multiple assertions

> `(assert-list` *object attribute*`)`
>
> Asserts a fact for each field of a multi-field attribute.

B Appendix: User function protocol.

This protocol must be followed when writing C functions which are to be invoked from the object oriented extension as Object methods. This includes procedures attached to attributes which are called `when-set` or `when-read`.

B.1 Overview.

Object methods are implemented as user defined C functions as specified in [2]. Information pertinent to the object oriented extension is provided via function call parameters.

The interface functions provided by CLIPS remain accessible. This includes the CLIPS parameter passing routines such as runknow.

Utility functions are provided for accessing and manipulating objects and attributes from user functions. These are used to access the invoking object or any named object.

`when-set` attached procedures are called before the attribute is updated. The new value is passed as the first CLIPS function parameter and is obtained using CLIPS interface routines. It is the responsibility of the function to update the attribute, possibly with a modified value.

All `WORD` or `STRING` valued parameters are represented by CLIPS hash pointers.

As well as including required CLIPS header files the source should include `objects.h` in order to access structures relating to the object oriented extension.

B.2 C Function parameters.

The object oriented extension calls methods and attached procedures with four parameters:

1. The name of the object which invoked the function. Type `HASH_PTR`.
2. The name of the attribute to which the the function is attached. `NULL` if not an attached function. Type `HASH_PTR`.
3. A pointer to the attributes data field. The field is a union of `float`, `HASH_PTR`, or `MULDATUM`.
4. The type of the data. An integer with possible values: `WORD`, `NUMBER`, `STRING`, or `MULTIPLE` as defined in `constant.h`.
5. The class from which the function was inherited.

The provided names allow for object specific and attribute specific processing. They can be used as parameters to object access utility routines.

The data field pointer can be used to access and update the attribute's value. If a `when-set` procedure the field will contain the previous value. The new value is obtained as the CLIPS parameter selected by parameter four. The `when-set` procedure is responsible for updating the value in the data field.

If the attribute is multi-valued the data field is a pointer to a linked list of the form:

```
typedef struct muldatum_fmt *  MULDATUM;

struct muldatum_fmt
{
        int data_type;          /* constant.h VALUE  */
        union {HASH_PTR symbolic; float numeric} data;
        struct muldatum_fmt *next;  /* LIST LINK        */
};
```

B.3 C Function returns.

when-read attached procedures return a resultant value which is reported as the attribute value. The procedure must be declared as an appropriate type and this type must be specified in usrfuncs.

B.4 Utility routines.

Certain object oriented commands are accessible to external C functions via function calls. The function names consist of the command prefixed by uf_ for "user function".

Parameters common to many routines:

Object identifier As with CLIPS function.

Attribute identifier As with CLIPS function.

type An int code indicating the type of an attribute value being affected. Can take the value NUMBER, WORD, STRING or MULTIPLE as defined in the file constant.h.

numeric A *float* attribute value.

alpha A HASH_PTR addressing either a CLIPS WORD or STRING attribute value.

B.5 Instance manipulation.

Creation

(uf_instantiate *class instance*)

Removal

(uf_delete_instance *instance*)

B.6 Attribute manipulation.

Setting values

uf_set_object_attribute(*object attribute [data [parameters]]*)
uf_append_to_object_attribute(*object attribute [data [parameters]]*)

Retrieving values

```
(get_object_attribute object attribute [data [parameters]])
```

B.7 Predicate functions.

The truth of predicates is returned as CLIPS_TRUE or CLIPS_FALSE as defined in constant.h.

Attribute of object?

```
uf_is_attribute( object attribute)
uf_is_attribute_numberp( object attribute)
uf_is_attribute_wordp( object attribute)
uf_is_attribute_stringp( object attribute)
```

Inheritance?

```
uf_inherits( class object)
```

B.8 Fact assertion

Single assertion

```
uf_assert_attribute( object attribute)
uf_assert_instance( object)
```

Multiple assertions

```
uf_assert_list( object attribute)
```

References

[1] Giarratano, Joseph C. *CLIPS User's Guide.* COSMIC, The University of Georgia, 382 East Broad Street, Athens GA 30602.

[2] Artificial Intelligence Section, Lyndon B. Johnson Space Centre. *CLIPS Reference Manual.*

[3] Artificial Intelligence Section, Lyndon B. Johnson Space Centre. *CLIPS Architecture Manual.*