Using *this-method* to Plan and Execute Tasks in CLOS

Martin Boyer
Robotics Laboratory
Hydro-Québec Research Institute
1800 montée Ste-Julie
Varennes, QC, Canada, J3X 1S1
mboyer@ireq-robot.hydro.qc.ca

Laeeque K. Daneshmend, Vincent Hayward
McGill Research Center for Intelligent Machines
McGill University
3480 University Street
Montreal, QC, Canada, H3A 2A7
laeeque@McRCIM.McGill.CA

1 Introduction

Advanced supervised/autonomous robotic systems may be characterized as large and long lived software environments, distributed over a large number of mechanical and computer subsystems. Appropriate hierarchical representations of such robotic systems are desirable, to facilitate system specification, implementation, and evolution. In the end product, this includes planning and programming of robotic tasks and recovery procedures, when tasks fail.

Strictly hierarchical designs of robotic systems, which rigidly enforce simple inheritance, have been notoriously unsuccessful in the past. Since the operation of such a system may be viewed from so many differing perspectives, simple inheritance is much too restrictive a representation tool. In this paper, we start by contrasting the traditional class hierarchies with the requirements of robot task planning and execution. A novel object-oriented design methodology is formulated, which satisfies the requirements of designing and implementing complex robotic systems. A short example of the results of this methodology is presented, followed by suggestions for future work.

2 The Method Hierarchies

Traditionally, software objects are used to model physical entities, where slot values represent states or properties of objects. Methods are then applied to one or more objects to change states and properties. Is it possible and sensible, on the other hand, to represent processes as objects? Imagine a machine shop,

and a production engineer who wants to organize and schedule the production of several, different, series of parts. There are common operations, such as cutting, drilling, deburring, and measuring, which are independent of the shape and purpose of the manufactured part. During production line planning and design, the actual steps and methods required to accomplish the common operations are less important than the prerequisites and results of the operations. More relevant are the raw materials, a smooth flow of operations, and the final product. The machinists can worry about manipulating the individual parts.

Using this model, the first task is not to model and represent the physical objects by CLOS objects, but rather to represent the operations as objects! Operations can have states and characteristics stored in slots. Cutting, for instance, requires a detailed drawing. Measurements must be done under supervision of the QA engineer. Machining may require special skills and tools for harder materials or better precision[1]. Again, to the people planning the operations, allocation of resources and flow of operations is more important than the actual low-level manipulations.

2.1 Reasoning about Methods: Planning

From this point of view, planning is seen as reasoning about processes, or methods. Further, error recovery can be thought of as local planning. In other words, to prepare a task or recover from execution errors requires some amount of reasoning about the actions that compose the task or the plan. It soon becomes imperative to organize the actions to be able to find common points between some actions, as one of the ways to reduce the resources required to describe those actions.

Some criteria must also be chosen to characterize the actions; to provide parameters to compare and "commonize" the actions and to provide "substance" for the planner and error recovery. Once common points have been identified, actions can be placed in a hierarchy in which complex actions, for instance, inherit characteristics from simpler, low-level, actions. The actions can thus be characterized and classified with respect to several parameters. This sort of characterization is essential in order to work out generalized planning and recovery strategies.

Among these characteristics are preconditions and post-conditions, which describe the state of the world prior and after execution of an action, but also the complexity of the action, how the action was defined, under what type of control, etc. All these parameters are intended to be used at planning time to help generating and validating a plan and at ex-

[1]While the skills and tools can be modeled as characteristics of the object, greater precision is largely due to the process.

ecution time to assist the operator, by providing explanation of the intent of the plan and preventing conceptual errors during interactive use of actions (e.g. when the operator has manual control over the system). Further, these parameters can be used to reason about actions and infer more preconditions and failure modes. Some useful parameters are:

- Types of actions:
  Initialization, Decision, Sensing, Motion, Organizational Change.
- Complexity Levels:
  Atomic, Low, Medium, High, Plan.
- Confidence Level:
  Simple, Routine, Frequent, Verified, Complex, New, Gamble.
- Source:
  Predefined, Run-time, Random.
- Control:
  Autonomous, Shared, Filtered-operator, Direct-operator, Physics.
- Actors:
  Manipulator, Tools, Sensors, Operator.
- Objects:
  Insulators, Crossarms, Nuts and bolts.

2.2 Representing Actions in CLOS

Using frames to describe actions at planning and recovery time, the above parameters become slot names in the action frames, allowing the corresponding slot values. Described as a CLOS class, the actions take this form:

```
(defclass plannable-action-mixin
   ()

 ;; These slots hold everything required to use an action in a plan.
 ;; Note that some slots are "advertised" as read-only;
 ;; the corresponding slot-values are set at compile time.

 ((type       :reader   action-type
              :initarg  :type)                ;computation/energy transfer
  (complexity :accessor action-complexity
              :initarg  :complexity)          ;atomic/plan
  (confidence :accessor action-confidence-level
              :initarg  :confidence-level)    ;routine/unverified
  (source     :accessor action-source
              :initarg  :source)              ;predefined/random
  (control    :accessor action-control
              :initarg  :control)             ;autonomous/human control

  (preconds   :accessor preconditions
```

```
              :initarg  :preconditions
              :initform ())
  (postconds  :accessor postconditions
              :initarg  :postconditions
              :initform ())
 )
)
```

The slots in this class have a one to one correspondence with the action parameters described earlier, except for the *Actor* and *Objects* parameters which are best determined at run time, when the action is actually performed.

A trivial example, defining the `move` class of actions, its generic function and a specialized method, can roughly be expressed in CLOS code as follows:

```
(defclass move
   (motion)                  ;MOTION itself inherits from
                             ; standard-method and plannable-action-mixin

   ()

   (:default-initargs
       :complexity           'low
       :confidence-level     'frequent
       :source               'run-time
       :control              'autonomous)
   )


(defgeneric move (object destination)
   (:method-class 'move))


(defmethod move ((object heavy-object)
                 destination)
   (...))                    ;take special precautions when moving a heavy object


(let ((method *))
    (setf (action-complexity method)    'medium)
    (setf (action-control method)       'shared))
```

3 Combining the Two Hierarchies

There are a multiplicity of software design methodologies which have been developed over the years. They are typified by focusing on one or another aspect of the overall software design problem. Hence, such methodologies are only effective in situations where other aspects of the design problem are so trivial that they can be handled in an ad hoc manner [San89].

The standard methodology for object-oriented design is to identify the relevant physical objects and map them into object classes, and then to specify the details of each object, including the methods associated with it. However, this methodology does not address the asynchronous, distributed, concurrent, nature of the problem.

In contrast, conventional "structured development", based on a top-down philosophy, maps physical operations to communicating concurrent computational processes. This decomposition is useful for coping with distribution and concurrency, but does not have the power of the object-oriented approach in terms of software modularity and reconfigurability.

Hence a fusion of the two approaches seems appropriate [Jal89]: to our knowledge, no such methodology has been postulated elsewhere. A standard hierarchical object-oriented design (or HOOD) design methodology, as applied to complex robotic systems, was developed previously at McGill [HDFB88] [HDF+90]. The HOOD paradigm supports a taxonomy of objects, but not does not cater for a taxonomy of the operations which act upon those objects. If we extend the HOOD paradigm by specifying that:

- operations on objects have an associated data type
- operation types inherit attributes from operation supertypes

we arrive at the Dual-Hierarchical Object-Oriented Design (DHOOD) paradigm. The DHOOD methodology utilizes the two taxonomies in the following manner:

1. Identify Physical Objects and Attributes
2. Map physical Objects to software object classes
3. Identify Physical Processes/Operations on Objects
4. Map Physical Processes to software processes
5. Associate software processes with software objects
6. Establish interface to objects
7. Implement operations (methods) on objects
8. Refine objects if they are too complex to implement, and repeat procedure starting from step 3; else terminate refinement.

Relating this approach to the design problem at hand, the following design methodology results:

i Formulate execution taxonomy: physical objects and their relations

ii Define methods for objects in execution taxonomy: correspond to physical operations
iii Formulate planning taxonomy: classification of methods in execution taxonomy
iv Define methods for objects in planning taxonomy: correspond to recovery plans for relevant methods
v Refine objects in execution taxonomy, if necessary, and repeat from step ii.

This methodology also resolves the problem of relating the two taxonomies in a consistent manner, provided that the above design procedure is followed.

3.1 Overloading Methods to Describe and Specify Actions

CLOS has been said to be self-contained; only a few basic classes and objects are created beforehand and the rest of CLOS is defined in terms of these basic elements. One of the benefits of this is the fact that methods are also objects (instances of the class `standard-method`) and hence can be manipulated within the CLOS framework. Thus, the same paradigm and tools can be used to perform run-time execution and off-line planning and reasoning. Further, COMMON LISP supports *facets* [CW85]; a single symbol can refer to multiple concepts and, in this particular case, to both an executable function and to a class of such functions. In practice, one can use the same "name" to *execute* a piece of code and to *reason* about this code. Accordingly, the action parameters turn out to be slots in method objects, as stated previously.

4 A Sample Implementation

This section outlines the VICTORIA DAY PCL source code modifications that were required to implement the connection between run-time code and planning strategies. These modifications are in no way complete and certainly not efficient, but rather, as we say, a *hack*[2].

Note that the current method object can also be obtained by a call to `(compute-applicable-methods generic-function arglist)` but this form has the significant drawback that it is slow; it recomputes the method from scratch, which seems unfortunate.

Rather, the concept is to create a special variable, `*this-method*`, and bind this vari-

[2]hack: 1. n. Originally a quick job that produces what is needed, but not well. *The Jargon File*, ©Eric S. Raymond

able to the method *object* as it is being executed. This requires changes to the method function constructor to declare the variable as `special` (which makes it dynamic [Ste84, p. 157]), and simply bind it to the "current" method. In essence, the executable code retains a pointer to its definition. The way it is done makes the variable `*this-method*` available in every method in the world without having to declare the method combination type or the class of the generic function (this is good since these capabilities were not available in VICTORIA DAY PCL). As far as we could determine, the variable is correct under these conditions:

1. Using standard method combination
2. Not using `:before`, `:after`, or `:around` methods

Two functions need to be modified: `make-effective-method-function` and `add-lexical-functions-to-method-lambda`. Both functions must extract and store the current method from the argument list. Additionally, the former must be modified so that the body of the method function holds (in `*next-methods*`) the list of the applicable methods, instead of a list of applicable method *functions*. That is, the following calls:

```
(let ((next-method-functions
       (mapcar #'method-function
               (caddr form))))
  ...
  (let ((*next-methods*
         next-method-functions))
```

are replaced by:

```
(let ((*next-methods*
       (caddr form))
      ...
```

And then, in `add-lexical-functions-to-method-lambda`, this call:

```
(apply .next-method. cnm-args)
```

is replaced by:

```
(let ((*this-method*
       .next-method.))
  ...
  (apply (method-function
          .next-method.)
         cnm-args)
```

5 A Test Case: Telerobotics

Consider a system consisting of a robot, an operator, and the various interfaces to allow the operator to control the manipulator directly or issue complex commands from a certain distance from the work area. In particular, this system is to be used to maintain live electric distribution equipment, as described in [Gir88] and [BDHF91].

The hierarchies of objects and actions for this task are depicted in figures 1 and 2.

This work has also been taken further in [PBD91] to include, for instance, trajectory

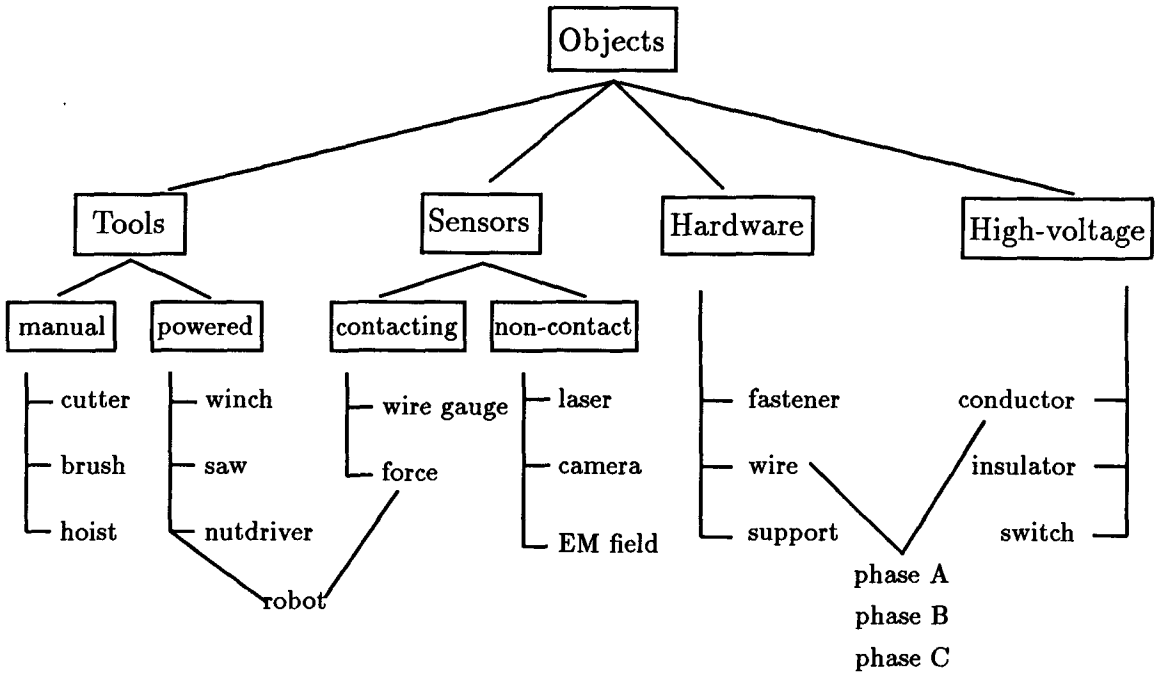

Figure 1: **The Objects Hierarchy**

control. Another test case is described in [HDFB88].

6 Conclusion

We have shown how processes can be modeled as CLOS objects, with slots holding properties, such as preconditions, essential for planning systems. Physical objects, on the other hand, have always been represented as CLOS objects, as is traditional is object-oriented programming languages.

We have also shown that it is relatively simple to implement an extension to CLOS to give access to the definition of a method from its executable component[3]. Indeed, certain implementations of COMMON LISP already provide such an extension, albeit not documented and not complete[4]. Further, we stated which properties of the LISP language are useful or even essential to the integration of the two hierarchies that are formed by the objects and the actions in a task.

[3]Strictly speaking, the executable part is in the generic function.

[4]LUCID's `wizard.doc` file, in the 4.0 release, hints that `apply-method` could be used.

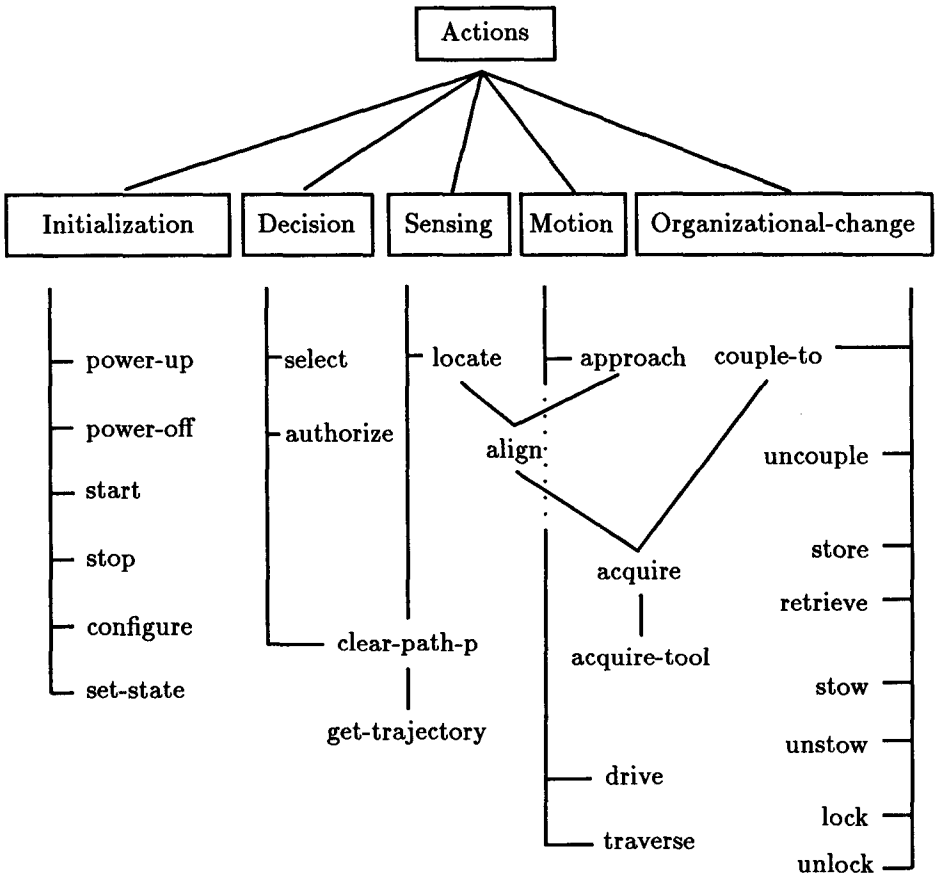

Figure 2: **The Actions Hierarchy**

Obviously, there are other ways to connect the two hierarchies (such as a separate set of objects for the representation of actions), but an integration of both, in the same programming environment, has tremendous advantages in terms of programmer productivity and self-documentation. If, then, CLOS is to be used as a basis for an integrated planning/execution system (an integration required for on-line error detection and recovery), there *must* be a relatively standard and painless way to access a method description from the executable code. In clear, we need a stable, documented, and simple implementation of `*this-method*`, where `*this-method*` is defined, in the body of a method function, as the method itself.

References

[BDHF91] Martin Boyer, Laeeque Khan Daneshmend, Vincent Hayward, and André Foisy. An object-oriented paradigm for the design and implementation of robot planning and programming systems. In *International Conference on Robotics and Automation*, pages 204–209, Sacramento, CA, April 1991. IEEE.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Gir88] Pierre Girard. La robotique en distribution. Technical report No. IREQ–4167C, Institut de recherche d'Hydro-Québec, Varennes, QC, Canada, April 1988.

[HDF+90] Vincent Hayward, Laeeque Khan Daneshmend, André Foisy, Martin Boyer, L. P. Demers, R. Ravindran, and T. Ng. The evolutionary design of MCPL, the mobile servicing station (MSS) command and programming language. In *International Workshop on Intelligent Robots and Systems: Toward A New Frontier of Applications*, Tsuchiura, Ibaraki, Japan, July 1990. IEEE. 1990.

[HDFB88] Vincent Hayward, Laeeque Khan Daneshmend, André Foisy, and Martin Boyer. Final report on the technology development for the MSS command and programming language (MCPL). Technical Report prepared under subcontract No. 79145TF, McGill Research Centre for Intelligent Machines, McGill University, May 1988.

[Jal89] P. Jalote. Functional refinement and nested objects for object-oriented design. *IEEE Transactions on Software Engineering*, 15(3):264–270, March 1989.

[PBD91] Michel Pelletier, Martin Boyer, and Laeeque Khan Daneshmend. A taxonomy for objects and actions in intelligent control of telerobots. In *Canadian Conference on Electrical and Computer Engineering*, pages 65.4.1–65.4.1, Québec, Canada, September 1991.

[San89] B. Sanden. The case for eclectic design of real-time software. *IEEE Transactions on Software Engineering*, 15(3):360–362, March 1989.

[Ste84] Guy L. Steele. *Common Lisp, the Language.* Digital Press, Burlington, MA, 1984.