Modelling the Robot Problem Solving Using the Set Data Structure

Giancarlo Colla¹, Giancarlo Succi² and Marco Ronchetti²
E-mails: elia@dist.unige.it, charm@dist.unige.it, marco@dist.unige.it
¹ DIST-Universita di Genova
Via Opera Pia 1 1a, I-16145 Genova, Italy
Phone: +39-10-3532997 - Fax: +39-10-353
² LII/DISA - Universita* di Trento
Via F. Zeri 8, 38068 Rovereto (Trento) - ITALY
Phone: +39-464-443136

Abstract
The robot problem solving is an important and significant argument because it is used in many environments: automated factories, automated vehicles and, in general, many artificial intelligence problems. The goal of the robot problem solving is to compute a sequence of actions that change the initial system configuration into a well defined final one. In this paper, it is described a new approach to the robot problem solver based on the set data structure. We have used SEL, a set-based language, to write a program that implement the model. Besides, the process and data parallel implementations of the robot problem solving are discussed.

1 Introduction
In many cases a process that handles an environment like an automated vehicles, a factory or a robot must act on it to achieve specific configurations. For example, it can park vehicles, it can control the execution of the factory machines or the movements of a robot hand. Generally, the controller knows the current state of the environment and the state that has to be achieved. Besides, it can act on the system by means of a group of actions. The problem solver is a fundamental element of the process because it computes the sequence of actions that, starting from the initial configuration, produce the final state. In this paper, it is discussed a set-based approach to the robot problem solving. In the first section the robot problem is introduced and its main topics are identified. It is also described the trivial and famous "black world" problem: it is used to explain the solving strategy based on the set data structure. The second section describes and analyzes the Prolog approach to the robot problem. The fourth section describes the Subset Equational Language (SEL), i.e. a programming language based on the set data structure that is used to implement the robot problem solving. In the remaining part of the paper the set-based approach is described and implemented. The advantages that the data parallel implementation of SEL are also discussed in this section.

2 The Robot Problem Solving
In this section the robot problem solving is introduced. It is also described the "black world" problem (ICW) that is used as an example to show some different approaches to the general problem. Even if this paper describes mainly the solution of this specific and trivial example, the adopted strategies can be applied to all the problems having the same structure.

The goal of a robot problem solving is to compute a sequence of actions that lead the state of the system into a specific configuration. A typical approach to solve this problem consists of identifying some elements: the state description, the goal and the actions that the robot can execute. The state description is used to outline the environment surrounding a robot and is often described using predicate calculus well formed formulas. As an example, consider the figure 1. It depicts a simple, but significative situation: three blocks are placed on a table and a robot hand acts on them.

The following formulas describe the initial state:

Clear(c) Clear(b)
OnTable(a) OnTable(b)
HandEmpty On(c,a)
Clear(c) means that the block c has no other block onto it, OnTable(a) means that the block A is placed directly on the table, On(c, a) means that the block c is placed on the block a and the state of the robot hand is specified by the HandEmpty formula. However, it is important to describe not only the current state, but also the final state that the robot actions must produce: the final state is often called goal. Well formed formulas can be also used to outline a goal. The final configuration that the robot hand of figure 1 must produce is the following: On(b, c) and On(a, b)). The robot constructs a stack of blocks in which the block b is on block c and block a is on block b.

An other important element is the group of actions that the robot can perform. In the block world example, the robot hand can pick up a block from the table or from the head of a blocks stack. In the same way, the robot hand can put down a block on the table or onto another block. Obviously, the robot can execute an action only if some preconditions are verified, e.g. the robot hand can pick up the block a only if the hand is empty and the block a has no other block onto it. Once an action is executed, the configuration of the "block world" is changed and, as a consequence, also the state description must be up-to-date: some formulas are no longer valid, while others must be added to it.

The previous considerations suggest a model to define a robot action that consists of three main components: the precondition formula, the delete list and the add formulas. As an example, consider the pick-up action by which the robot hand picks up a block:

```
pickup(X):

Precondition:
OnTable(X) and HandEmpty and Clear(X)

Delete List:
OnTable(X), HandEmpty, Clear(X)

Add List:
Holding(X)
```

The precondition asserts that the hand can catch a generic block X only if X is on the table, the hand is empty and X has no blocks on its. The delete list and the add list specifies that the action change the state description by deleting from it the OnTable, HandEmpty and Clear formulas and adding to it the Holding one.

Once the goal, the state description and the actions have been modelled, the problem solving strategy must produce a sequence of actions that change the system state into the one described in the goal. The mechanism is comparable to a tree search. In fact, at each step, the system identifies a group of actions that are executable using the precondition list and, then, produces as many new states as the number of executable actions is. The main steps that are needed to scan the execution tree are:

* preconditions analysis: the precondition formulas bound to every action can perform are compared with the state description. This phase identifies both a set of actions that are executable and the objects on which they act.

* new state generation: for every allowed action, the problem solving mechanism produces a new state by means of the delete list and of the add formulas.

* pruning: according to pruning rules some branches of the tree are cut (e.g. if the new state is already present in a different node of the graph).

* goal checking: in the last phase the system checks if the goal has been achieved.

* recursion: if the goal has not been reached the problem solver re-executes all these phases using the new states that the last recursion level has produced.

Once the goal has been achieved, the problem solver runs through the execution tree starting from the root and reaching the leaf that corresponds to the goal. The crossed links identify the sequence of actions to be applied and the crossed leaves specify the sequence of configurations that the system will take.

3 Modelling Robot-Problem Solving

In the last section the most important features of the robot problem solving have been introduced. What has been discussed is only the conceptual model, however it is also important to describe the techniques that are used to implement it.

Numerous programmers implement the robot problem by means of a declarative language like Prolog ([CW]).

In this way, it is possible to describe the state of the system using lists of functions and to specify the actions by means of logical clauses. For example, the system configuration that is outlined by figure 1 can be identified by the following definition:

```
[OnTa(OnTb), Clear(c), Clear(b), On(a, b), HandEmpty]
```

The pickup action can be defined by the rule:

```
Pickup(X, New_L) :- Is_In_List(OnT(X), L), Is_In_List(Clear(X), L),
                  Is_In_List(HandEmpty),
                  Del_From_List(L, OnT(X), Clear(X), HandEmpty), New_L,
                  Add_To_List(New_L, Holding(X), New_L).
```

At the beginning of the execution, the L variable is bound to a ground term that defines the initial state of the system (in our example [OnT(a), OnT(b), Clear(c), Clear(b), On(a, b), HandEmpty]). The X and New_L variables are unbound but, at the the end of the clause execution, they will contain respectively a reference to the object on which the pickup action is applied and the new state description that it produces.

The three Is_In_List definitions implement the precondition analysis and bind the X variable to an element that satisfies all the preconditions (if this element does not exist the pickup action fails). The first Is_In_List identifies an element contained in the list that matches the OnT functor (in the first step of our example it is the a block). In this way the variable X is bound to the a term ground. The second Is_In_List verifies that Clear (c) is included into the list. Because it is not true, the block a is not suited to be picked up. As a consequence, the Prolog execution backtracks and the first Is_In_List is again processed. The Prolog machine looks for a second occurrence of the OnT functor and binds the X variable to its argument; in this way the new value of X is b. The second and third Is_In_List check respectively if the Clear(b) and the Handempty functions are contained into the list. Because the block b verifies all the preconditions, it is picked up and the state described by the list L must be changed.

The state change is performed by the Del_From_List and Add_To_List clauses that model the add and delete lists described in the preceding section. At the end, the new state is defined by the list that is bound to the New_L variable and the element that is involved in the operation is identified by the X one.

It is possible to declare all the actions that the robot can execute in a way similar to the pickup definition (e.g. the pushdown operation by which the robot puts a block on the table, the stack operation by which the robot puts a block onto a block stack and the unstack operation that defines the operation of picking up a block from a head of a stack). Therefore, N Prolog clauses are needed to specify N actions, one for every action.

Once the actions are declared, it is necessary to define the "plan generator" that produces all the new states from the beginning:

```
Prob1_Solv(State(Act, St, L), State(State(Actions "pickup", [l] [Act], New St)) := Pickup(St, L, New St).
Prob1_Solv(State(Act, St, L), State(Actions..., Pathdown(St, L, New St)).
Prob1_Solv(... Stack(St, L, New St)).
Comp_Goal(Start St, Goal, St, []) :- Is_In_List(St, Goal, St). Start_
Comp_Goal(Start_S, Goal_S, Act_List) :-
  Search_Tree(State[], Start_S, Goal_S, Actions, []).

Search_Tree(Curr_S, Goal, Actions, Glok_Tree) :-
  Is_In_List(Gol, Curr_S).

Search_Tree(Curr_S, Goal, Actions, Old_Tee) :-
  Probl_Solv(Curr_S, New_S).
  Pruning(Old_Tee, New_S).
  append(New_S, Old_Tee, New_Tee).
  // The tree is pruned
  // The new state is generated
  // The list of the visited nodes is built

Pruning(L1,L2) :- Is_In_List(L2,L1), fail.

Pruning(L1,L2) :- Add_To_List(L1,L2).

A description of the system state is defined not only by a list of functors that settle
the relations among objects (e.g. Om(a,b), ...), but also by the sequence of actions (e.g. pickup(a), ...)
that have generated the new state from the starting one. For this reason, the system state
is stored into a functor having two different arguments: the first is a list that contains the actions sequence that has been specified by the
second argument.

To identify the actions sequence that generates the final configuration of figure 1 starting from the beginning
configuration that is shown in the same figure, the Prolog program must be queried in the following way:

?-Comp_Goal([Clear(b), Clear(c), Om(a)), HandEmpty, OmT(a), OmT(b)],
   [On(a,b), Om(b), Actions]).

The Comp_Goal query passes the initial state as the first argument and the target state as the second argument.
At the end of the prolog execution, the Actions variable will be unified with the correct actions sequence.
The clause checks if the starting and the final state corresponds: if this is true the actions list is an empty list, otherwise
(often the case of our example) the Search_Tree definition is executed.

The second definition of the Search_Tree verifies if the goal has been reached. The second definition develops the
real generation and the searching of the decision tree:

1. Probl_Solv produces a new state starting from the current one. After the execution of Probl_Solv the
   New_S variable is unified with a new State functor that defines both the new generated state, and the
   applied action. If other decisions are possible, the Prolog marks this clause as a choice point.

2. The Pruning clause ends the processing of a tree branch that does not lead to the solution. We have implemented a
   trivial pruning strategy: the system ends the processing of a tree branch if it meets a node that has been already processed in a
different portion of the tree. This analysis is very important, because not only it optimizes the execution time by avoiding
operations that have already been executed (different sequences of actions produce the same state), but also it avoids infinite loops
(am sequence actions that produces a state that is equal to the initial one).

3. Append adds to the global list of visited nodes the new one that has just unified with the New_S variable.

4. The Search_Tree is recursively executed until the goal is reached.

3.1 Considerations

It is possible to notice some important features on the Prolog approach:

- The problem solving mechanism is a tree search; in fact, starting from the first state, new states are
  produced applying executable actions. A state has as many sons as the number of executable actions in
  the child that connect the nodes symbolize the actions (decisions).

- The strategy that is implemented in this section is defined as "depth first search", because the system
  follows the branches of the tree until it finds the one that leads to the solution or until it is pruned. This
  approach is typical of Prolog systems, because the computing model of the language is based on
  the concepts of clause points and backtracking. Everytime a Prolog system finds a point where different
  choices can be applied (multiple definitions of a clause as for the Probl_Solv one), Prolog unify the
  unbound variables of the calling query with the first term grounds that verify the conditions. Then the
  Prolog machine marks this point as a choice point and continues the computation. If the execution path
  fails (i.e. the goal is not reached), Prolog backtracks to the last marked clause point and tries an alternate
  choice. This computing mechanism corresponds to a "depth first" search: if a tree branch does not lead to
  the correct node, the running through the tree is restarted taking a different path in the last encountered
  forkings.

- The pruning mechanism is fundamental, because it makes possible to cut all the branches of the tree
  that do not lead to the goal. This task is performed saving into a global list all the nodes that the Prolog
  program has already processed and using it to avoid the duplication of subtrees and the iterations on
  infinite loops.

These features suggest the following considerations:

- The "depth first" strategy identifies an actions sequence that leads the system state into the final configu-
  ration. However it is not possible to identify the shortest actions sequence without generating the all
decision tree.

- The pruning operation is computing expensive because it has a quadratic computing weight (if N is the
  number of the tree nodes, the number of comparisons that are needed is N * (N - 1)/2).

- The strategy is suited to be implemented on a process-parallel architecture ([1969]). In fact, different
  branches can be executed on different processors. However, a little slow down is induced by the global
  data area that is represented by the Glok_Tee list.

- The process-parallel implementation distributes the computing on all the processors of a parallel machine.
  However, the "only" advantage is obtained because the Prolog machine runs through different paths at
  the same time while other operations continue to be very expensive. The analysis of the preconditions list
  still requires a time that is proportional to the dimension of the state description (a list of elements).

- Besides the operation has to be repeated for every possible matching and for every different action. The
  operation of removing from the current state description the elements that are contained in the delete
  list requires again the scanning of all the elements belonging to the state list. The pruning operation has
  the same previous problems. Besides processes residing in different processors need to synchronize the read
  and write accesses to the list that contains all the processed nodes.

In this section, the Prolog approach has been outlined. However, it is possible to implement the robot
problem using other strategies, one of them is based on the set data structure. To implement the theoretical
set-based model we have used the Subset Equational Language (called SEL). We have used it not only
because it handles sets of elements, but also because it has been implemented on a data parallel machine (the
Connection Machine 2). This approach does not achieve the parallelism distributing the operation of running
through the tree on different processing elements, but exploits the data parallelism that is implicit in operations like
the one that we have described, the pruning strategy.

In the next section the SEL and its data parallel implementation is introduced.

4 The Background: SEL, SAM and CM2

This section gives a brief introduction to the framework in which this research has been developed. It presents
the set-based declarative language we use, the Subset Equational Language (SEL), the abstract machine we
have developed for its implementation, the Subset Abstract Machine (SAM), and the Connection Machine 2.
It is not a complete description of these topics, which can be found in other papers, but only a short overview
and readers who are already inside these topics can skip it.

SEL has been developed by Jayaraman et al. at UMC/Chapel Hill and at SUNY/Buffalo [1968]. This
language handles sets in a clean and simple way. A SEL program is a sequence of two kinds of assertions:

\[
\begin{align*}
  \text{f/terms} & = \text{expression}, \\
  \text{f/terms} & \text{ } 2 \text{ expression}.
\end{align*}
\]

(1) is an equational assertion.
(2) is a subset assertion.

The meaning of these assertions:
(1) the function $f$ applied to the ground instances of terms is equal to the corresponding ground instances of expression;

(2) the function $f$ applied to the ground instances of terms contains the corresponding ground instances of expression.

A SEL query does not contain any variable, differently from Prolog, but only ground terms. There is no mechanism for backtracking and unification since only matching is required. The language incorporates the collect-all assumption for subset assertions, which states that the result of a function application to ground terms is the union of all the subsets obtained by all the subset assertions matching the ground terms with all the possible matching. More detail about the language can be found also in [Suc91]. A few examples of SEL programs can help understanding SEL.

Suppose we want to compute the square of a given set: we can easily do it with

$$\text{squareSet}([x]) \text{ contains } \{x \times x\}.$$  

The result will be, by the collect-all assumption, the set containing the squares of all the elements. Here a remarkable feature of SEL is present, the multiple matching: a matching of the kind $[\{x\}]$ produces the matching of $x$ with all the elements of the argument set allowing to perform the same operation on each one since no order is imposed over the elements of a set. Moreover the result of this assertion, with the same argument set, will be the same whatever matching is chosen (it will be shown further that non-deterministic results may appear with particular functions, but it is a marginal event).

The assertion to compute the intersection of two sets can be stated as:

$$\text{set.intersect}([x],[y]) \text{ contains } \{x\}.$$  

where each element of the first set must be matched in second to obtain the final result. It will be shown that hash tables can improve performances in situations like this. A more complicated example is:

\[
\text{params}() = \{\}.
\]

\[
\text{params}([x]) \text{ contains distr(x, params())}.
\]

\[
\text{distr}(x, (\{\})) \text{ contains } ([x]).
\]

which determines all the possible permutations of the elements of a set. The computation proceeds generating all sets matching the pattern in linear time and then applying to all the sets distr.

The unordered structure of the elements of a set allows each application of an operator to an element of the set to be independent from the other ones; therefore the same operation can be performed on all the elements in parallel exploiting the intrinsic data parallelism of SEL [SDG+94, SSS81].

The intrinsic parallelism of logic programs can be divided in two different classes:

* process parallelism
* data parallelism

The former form of parallelism parallelizes the execution of independent parts of a program identifying code fragments which can be performed independently. It is necessary to decide whether or not to execute them scheduling those executed on processors and collecting the final result. This approach requires quite a lot of communications between processes, for synchronizing and exchanging data, and a lot of time to handle processes for having a consistent execution.

The latter approach to parallelism parallelizes the execution identifying in the program cluster of data on which to perform the same operation, distributing them on available processors and manipulating them in parallel as much as possible. It is quite appropriate to be implemented on SIMD machine and it overcomes most of the limitations of process parallelism. However it requires a language suitable for the representation of collection of objects like SEL [Suc92, SM92a].

In a sequential implementation the assertion 3 is performed in $n$ steps, being $n$ the number of elements of the set. It is indeed necessary to select the elements of the set one by one and compute the search of each one, collecting the final result. A data parallel implementation on a SIMD architecture, like the Connection Machine, can perform the same operation in just one shot: if the argument set is already distributed among the processors, what is needed is just to ask each processor to square the element stored on it and this can be done in parallel.

The implementation of SEL is divided in two phases:

* the development of a compiler targeted to an abstract machine.
* the implementation of the abstract machine on the real architecture.

The abstract machine is called SAM [SM92a] and belongs to the WAM [AN90] family because its general structure resembles quite a lot that of the WAM. Two different implementations of SAM have been developed until now:

a **serial implementation** for a monoprocessor architecture.

a **parallel implementation** for the Connection Machine

that use the same instructions set so that the same program, once compiled, is able to run on either version.

Figure 2 outlines the general structure of the SAM: in addition to the standard components of the sequential SAM, like heap, stack, push-down-list and processor, there is the Active Memory used in the parallel version, that is a memory whose cells both store data and perform computations. Since SAM does not perform full unification, there is no need of the "trail" like in WAM, and faster store1 and match instructions replace the unify ones.

The aim of the active memory is to hold the sets to exploit the data parallelism of certain operations that, otherwise, would be executed sequentially. Its structure is a multidimensional array of cells, which can be implemented easily on the Connection Machine, where to each cell is associated a processor together with a set of registers and memory. The memory of each cell is organized in two parts: a stack, for performing local computations, and a region for keeping set elements. It should be noted that also in the serial implementation sets are stored apart from the data because they are managed slightly differently.

SAM and sets. The SAM takes advantage of three main situation to exploit the data parallelism of problems:

* when there are foldings of a set in a single element.
* when there are mappings of one set into another.
* when there are filters applied to a set.

Foldings are definitions of the kind:

$$f(\{\}) = x.$$

$$f([x]) = x(f([x])).$$

Here it is possible to perform a tree-like computation in order to determine the result. Note that this operation is not deterministic, since no order is imposed on the elements of sets, therefore it is possible to obtain different results for the same operation in subsequent executions. As a simple example the function nonDet

\[
\text{nonDet}([\{\}) = 0.
\]

1The store instructions store data, like lists, structures and simple constant data on the heap while all those data being elements (or part of element) of sets are put in a different region.
applied to the set \{1, 2, 3\} can give 0, 2 or 4 as result, depending on which matching is chosen. However [MS89]
demonstrated that if the folding function is commutative and associative the result is the same no matter of
the matching. Using an associative and commutative function like plus in
$$\text{det}(\emptyset) = 0.$$
$$\text{det}(\{x\}) = \text{plus}(\text{det}(\emptyset), \text{det}(\{x\})),$$
and applying det to \{1, 2, 3\}, the result is always 6.
The mapping operations are those which iterate over the elements of a set having the aim of generating a
new set whose elements are functions of the elements of the original one.
They can be divided in different categories, depending on the space needed to perform the matching process.
In the pattern
$$f(\{x\}) \text{ contains } g(\{\}).$$
we need constant space since we just scan the set. Instead with
$$f(\{x\}) \text{ contains } g(\{x, T\}).$$
we need quadratic space, since we need to build \(n\) copies of the set without one element, being \(n\) the cardinality
of the set. In a data parallel environment the mapping of a set into another is an operation that can be
performed in one shot. Given the theoretic definition of set2:
$$\text{set2} = \{f(x) : x \in \text{set1}\}$$
it is possible to compute set2 applying \(f\) to all the elements of set1 in parallel. In figure 1 is shown an example
using square as mapping function.

![Diagram](image)

**Figure 3:** Mapping set1 through square to obtain set2.

Filtering instructions build a new set with the elements of their set arguments that satisfy a given predicate.
An example is given by the definition of set_filter:
$$\text{set_filter} = \{x : x \in \text{set.glob}, p(x)\}$$
where set_filter is the set containing those elements of set.glob that satisfy \(p\).
Again it is possible to have a data parallel implementation in just one shot. This is performed disabling those
cells of the active memory where the predicate \(p\) is false.

The Connection Machine System  The Connection Machine System is designed for high-speed data parallel
computation[7].
In conventional computing a computer has a single central processor which operates on data sequentially. If
the same operation is to be performed on many data the computer must perform the operation separately on
each element one by one.
In data parallel computing there are many processors and it is possible to associate each data with one of
them so that the same operation can be performed by all processors, at the same time, on all data. This

---

1 We will refer to Connection Machine also as CM

paragraph gives a brief description of the CM's architecture and of few of its facilities.
All CM Systems contain a parallel processing unit, at least one front-end computer and a high-performance
parallel I/O system. The parallel processing unit, illustrated in figure 4, is in the heart of the CM system. It contains
thousands of individual processors with an ALU and up to 64 Kbits of memory of their own. The processors are
controlled by a device called sequencer whose job is to decode and broadcast commands to the processors for
parallel execution (CMs have up to four sequencers). Processors can be organized in multidimensional grids,
called shapes, used as template for parallel data. Each shape is defined by
- the number of dimensions,
- the number of positions in each of its dimensions.

It is then possible to define variables of a defined shape, parallel variables, on which to perform parallel
computation. The difference between a scalar and a parallel variable is that the former can contain only one 'item'—e.g. one integer, one character, and so on— but the latter can contain more than one since it is associated to a shape.
Therefore the global number of item in a parallel variable is equal to the total number of position in its
shape, which is the product of positions in each dimension of the shape.
The processors are interconnected by a high-speed communication device called router which allows to send
or to receive data from other processors in parallel, or to exchange values between parallel variables. The
parallel processing unit also supports a faster form of communication, called grid communication or NEWS
communication, which allows processors to communicate with their neighbors in a multidimensional grid.

In the latter type of communication elements of parallel variables in the same shape communicate in regular
patterns by using their coordinates, that is values of all elements in a parallel variable move the same number
of positions in the same direction. In the former any parallel variable element can communicate with any other,
whether or not they are of the same shape, and whether or not the pattern of communication is regular. This
is more versatile than grid communication but it is also slower. The parallel processing unit appears to the user
as an extension of the normal environment of a standard serial computer, referred to as a front end, the user's
gateway to the CM system.

Of course there are parts of a program which can be executed in a more efficient way with the classic serial
approach that can take place on the front-end machine. Multiple front-end computers can be connected to a
single parallel processing unit through a switch called ares. When programming in data parallel, organizing
data so that the same operation can be carried out on many elements at once, it is possible to have more data
than processors. If it is the case, the CM system creates virtual processors by dividing up the memory associated
to each physical processor. This allows the user to write programs assuming he can use as many processors
as he needs. The same program can run, without change, on different parallel processing units with different
numbers of physical processors.
Since there are few differences between programs written in data parallel languages and classical sequential
programs, it has been more profitable to add new syntactic forms to well-known languages instead of creating
new ones. Therefore we can program on the Connection Machine with high-level languages such as C*, a
data parallel extension of the C programming language, CM Fortran and *Liap, data parallel extensions of
Common LISP and Fortran 77. In addition a low-level parallel instruction set called Parris is provided.
5 Modelling Robot Problem Solving Using Sets

Section 3 describes the implementation of the block world problem using a declarative language like Prolog. In this section a different approach, based on the set data structure ([Jay90b]), is discussed. The "block world" problem is used as an example to explain the main topics of this solution. However the method that is described can be applied to the most part of the robot and planning problems.

5.1 The State Description and the Objects

Figure 5 outlines the fundamental steps that are needed to solve a robot problem. First of all, it is important to define the elements involved in the problem; this task is performed by a set containing the term groups that identify all the blocks that the robot must move. The following SEL assertion defines the set (JNB88):

\[
\text{elem.set} = \{a, b, c\}
\]

Once the elements are defined, it is necessary to describe the relations that exist among them (e.g. the block \(a\) is on \(b\), the robot hand is empty and so on). This is achieved by means of a second set that contains a collection of functions:

- \(\text{onT}/2\): e.g. the onT(a) functor specifies that the block \(a\) is on the table,
- \(\text{handempty}/1\): e.g. the handempty asserts that the robot hand is empty,
- \(\text{holding}/1\): e.g. the holding(b) asserts that the block \(b\) is held by the robot,
- \(\text{on}/2\): e.g. on(a,b) declares that the block \(a\) is onto the block \(b\),
- \(\text{clear}/2\): e.g. clear(c) asserts that no other block is on \(c\).

A collection of these functors describes the state of the world-block.

However, the sets that define the blocks and the relations among them are not sufficient to describe the world state for a robot problem solving. In fact, a single set gives only a snapshot of the block world, while it is necessary to define also its evolution starting from the the beginning configuration. This goal is achieved considering the sequence of actions that has produced every single object relation.

5.2 The State Tree Representation

The problem solving can be compared to a tree search. In fact, the problem solving strategy tries to identify an actions sequence that leads to the goal applying all the executable actions to every system state (see figure 6). As first, the problem solver identifies all the actions that can be performed on the beginning configuration, and, subsequently, produces the new states using the delete list and the add list that are bound to every action (layer 2 in figure 6). If the goal has not been achieved, the problem solver re-executes the preceding steps to all the tree leaves that it has produced in the last phase. Therefore, it is very important to describe not only the "block world" state, but also its location into the "states tree". The information that is saved in addition to the state description must identify both the sequence of actions that has produced the current state, and its depth into the tree (second layer, third layer,...). The actions sequence is bound to every object relation description (e.g. on(a, b)) by means of a functor state/2. It has as second argument the actions list that has generated the object relation placed as first argument of the functor. SEL models the actions sequence as a list of functions name.action/2, each one identifying both the action performed by the robot and the elements on which the action has been executed. For example, the SEL declaration stack(a,b) identifies the action stack that is applied on the elements \(a\) and \(b\). It is important to notice that every layer of the decision tree is identified by a collection of functor state/2. The second argument of them is used to distinguish the elements belonging to the different states that can be generated with the same number of actions.

5.3 Modelling Robot Actions

At this point the actions must be defined. For example, the pickup action is declared by the following subset assertions:

- \(\text{PickUp(X)},\)
  - \(\text{state(onT(X)}, L)\), \(\text{state(holding(X)}, L)\), \(\text{state(holding(X)}, L, \text{add} \_ \text{List}, \text{Glob}\_\text{St}, \text{All LSTM Act})\)
  - \(\text{Gen}\_\text{New}\_\text{State}\_\text{Old} \_\text{List}\_\text{New} \_\text{List}, \text{Glob}\_\text{St})\)
  - \(\text{state(onT(X)}, L)\), \(\text{state(ready(X)}, L)\), \(\text{state(ready(X)}, L, \text{add} \_ \text{List}, \text{Glob}\_\text{St}, \text{All LSTM Act})\)
  - \(\text{Gen}\_\text{New}\_\text{State}\_\text{Old} \_\text{List}\_\text{New} \_\text{List}, \text{Glob}\_\text{St})\)

The arguments of the PickUp assertions are mainly declarations of set matching.

1. The first implicit iteration is performed on the objects that are involved in the robot problem (a, b, c,...),
2. The second set matching identifies all the states belonging to the previous layer on which the PickUp action can be executed (the different states are characterized by different actions list L).
3. OldList addresses all the state/2 structures that are contained in the previous layer.
4. GlobSt addresses all the state/2 structures that have been processed until now (it is used by the pruning mechanism).
5. All.List.Act addresses all the different sequence of actions that are bound to the GlobSt elements (it is used by the pruning mechanism).

Exploiting the Implicit Data-Parallelism of the Action Generation It is very important to notice that the precondition analysis is performed exploiting the data parallelism that is implicit in the operation (EM292). At the end of the matching operation of the second argument, the variable \(L\) is bound to a multiple data that identifies all the configurations of the previous layer on which the action is executable. If we are using the CM2 SEL implementation, these elements are distributed over the CM2 processors. Once the \(X\) variable is bound to an object (the set matching of the first argument), the abstract machine identifies a collection of actions
lists L for which the first precondition is true (onT(X)). This operation requires a single step. In a second time, the SEL executor verifies the other preconditions (clear(X) and handempty) performing the checking in parallel on all the processors. If N1 actions lists have matched the first precondition than the second checking is repeated N1 times, the number of the valid actions lists decreases to N0. The third checking is repeated N0 times. (Notice that L is only a little subset of the initial set). As a consequence the system identifies all the previous configurations on which an action can be executed in little more that a single step (figure 7).

Once the matching of the assertion head has been performed the variable L is bound to a multiple data that addresses all the configurations belonging to the previous layer on which the action is applicable. The SAM abstract machine iterates sequentially for each single element of L to build the new layer. The Del From and Add To assertion in the Gen.New.St execution produce the new state/2 functions that belong to the new layer. These two operations require a time that is linear with the dimensions of the add list and delete list (generally their dimensions are little). The global time that is required to produce the N0 new configurations (N0 is equal to dimension of L) is linear with the number of configurations on which the action is executable figure 7).

5.4 Pruning the Tree

The Pruning assertion cuts all the new nodes that are already present into the tree. This operation is very important, because, in this way, it is possible to save computing cycles. CM2 processors and time. In fact, if the actions generation is executed on two equal states the result is the same, i.e., two different subtree that have, as root, two equal states corresponds one another. If the pruning strategy is not implemented, some operations are useless repeated.

The next SEL assertions define the pruning operation:

\[
\text{Already}((0),...) = \text{true,} \\
\text{Already}((0), (\text{state}(X,0),0), (\text{Glob.St})) = \\
\quad \text{if Not.Empty(Intersection(Old, Prod.St(X, Glob.St)))} \\
\quad \text{Already(Prod.St(X, Glob.St))} \\
\quad \text{else false.}
\]

\[
\text{Prod.St(X, state}(X, L), L) \text{ contains (L).}
\]

\[
\text{Not.Empty}() = \text{false.} \\
\text{Not.Empty}((X)) = \text{true.}
\]

\[
\text{Intersection}((.,)) = \text{false.}
\]

Pruning(\text{New.St, Glob.St}) contains

\[
\quad \text{if Already((0), New.St, Glob.St))} \\
\quad \text{else New.St.}
\]

The Pruning assertion binds the new state that has been produced by the action execution to the New.St variable. The Glob.St variable is bound to the set that contains all the nodes belonging to the already processed tree. At this point, the assertion Already processes the two variables: it returns the value true if the new system configuration is already present into the tree, otherwise it returns the value false. In the first case, the Pruning assertion returns an empty set (and, as a consequence, no new nodes is added to the decision tree). In the second case, the new nodes are added to the decision tree.

It is interesting to notice how the Already assertion checks if the new set is already present into the tree (Glob.st).

Already takes one relation description (e.g., onT()0 at a time from the set that addresses the new configuration and binds it to the X variable. The remainder of the set is bound to the variable S.

The Prod.St assertion produces a set that contains a collection of actions sequences that have generated the state X.

The Intersection assertion has the following behaviour:

1. If the set generated by the Prod St assertion is empty, then there is no configuration in the tree that corresponds to the new state (first instance of Intersection).

2. If the first set is empty (it is the first execution of the assertion) then Intersection returns an empty set that contains a dummy element.

3. The Intersection returns the set that is produced by the intersection of the sets of the first and second argument.

The Not.Empty assertion is true if the set addresses by its argument is not empty, otherwise it is false.

The idea that this little SEL program implements is trivial, but functional. Taking one element at a time from the new state description the Prod.St generates the collection of actions lists belonging to the decision tree that have produced an equal relation. The operation is repeated on the next element of the new state description, creating in this way a new set that contains the actions lists bound to the new relation. The two sets (this and the previous one) are intersected generating the set that contains the actions lists that satisfy both the relations. This operation is repeated for all the elements in the new state. If the intersection of the actions lists set is empty, then no tree configurations correspond to the new one. Otherwise, if the final set is not empty, the current configuration is already present in the tree.

### Data-Parallel Execution of the Pruning Mechanism

The pruning strategy is very well suited to be implemented on a data-parallel architecture. In fact, the computing weight of the pruning assertion is \(O(N)\) where \(N\) is the dimension of the sift that describes the new state. \(N\) is only a little portion of the dimension of the decision tree. In the Prolog process parallel implementation, the pruning operation had not only a computing weight \(O(N)\) but also the pruning mechanism was necessary to avoid infinite loops and created synchronising problems. In the data parallel implementation the pruning does not waste many computing cycles.

The Prod.St and the Not.empty assertion requires a single data parallel operation. Instead, the Intersection assertion processes two different sets that contain the actions lists bound to the relations of the new state. These two sets have dimensions that are very smaller than the set that defines the decision tree. Besides, the dimensions of these sets decreases more and more after every new iteration of the Already assertion. The computing weight of the Intersection is \(O(N)\) where \(N\) is the smaller dimension of these two sets. The Already assertion executes recursively on the set that defines the new system configuration; it ends the execution only when the set has been processed. Therefore, the computing weight of the operation is related to the dimension of this set.

### 5.5 The Query

Until now we have defined the actions that the robot can apply to the system. Simply, these actions take nodes of the tree and produce new nodes. At this point, it is necessary to specify a new group of assertions that know only the initial and final configurations, control the generation of the tree.

The \text{IsGoal} declaration is used to start the processing.
Is_Goal(Start,Conf,Goal) contains
if Reached_Goal(Trans(Start,St),Goal) \{[]\}
else

Comp_Act_Seq()Trans(Start,St).Goal).

Trans(St,\{\}) contains (state(St,\{\})).

The \textit{Trans} assertion translates a set that contains the state description (e.g. state(A),on(T),...) into the first layer of decision tree. In fact, every element is a functor state/2. \textit{Reach_Goal} checks if the initial configuration is already equal to the final one; if this is not true, then the \textit{Comp_Act_Seq} assertion computes all the actions sequence needed to achieve the goal.

The \textit{Comp_Act_Seq} has three arguments:

1. It is bound to the set that contains a collection of elements, each one defining an object relation (e.g. state(A)) (see figure 6) that have already processed. The first time that the assertion is executed (layer 0), the variable is bound to an empty set.

2. The second argument of the assertion corresponds to a layer of the decision tree. The implicit iteration binds the \textit{Curr_St} variable to every configuration belonging to the layer.

3. The \textit{Goal} variable is bound to a set that defines the final configuration of the system.

The \textit{Reached_Goal} verifies if the goal has been reached. If the check has been successful then the state function that defines the actions sequence to achieve the goal is returned. Otherwise the \textit{Comp_Act_Seq} is recursively executed. The variable addressing the system state that is currently processed is added to the \textit{Global} and the new valid configurations are computed by the \textit{Apply_Act} assertion.

\textit{Comp_Act_Seq} contains
if Reached_Goal(Curr_St,Goal) RecAct(List(Curr_St,Goal)
else Comp_Act_Seq((Global_St,Curr_St),Apply_Act(Curr_St,Gal).

The \textit{Apply_Act} assertions take as the unique argument the current configuration of the system and applies to it all the possible actions: \textit{PickUp}, \textit{PutDown}, … . The \textit{Apply_Act} is defined by multiple instances, one for every action. The assertion returns a set containing all the new states that can be generated by the first one.

\textit{Apply_Act(Conf_Set)} contains
\textit{PickUp} \textit{(elem_set,Conf_Set)}.
\textit{Apply_Act(Conf_Set)} contains \textit{PutDown} \textit{(elem_set,Conf_Set)}.
\textit{Apply_Act} ............

\textit{Apply_Act}............

5.6 Considerations

The set data structure seems to be well suited to implement a problem like the "world-block" one. In fact, it is not only trivial to model the system and the actions, but also it has great advantages because the implicit data parallelism of the robot problem solving can be entirely exploited.

Acknowledgments

This work has been partly supported by the Italian ministry of university and scientific research (40% foundings). Most of the ideas presented in this paper has been discussed with B. Jayaraman. The authors thank researchers of DIT/Parma for their support in the usage of the Connection Machine.

References


