Artificial Intelligence in Engineering 13 (1999) 11–20

ARTIFICIAL INTELLIGENCE *in* ENGINEERING

A factorisation model of robotic tasks

Salima Benbernou

Institut d'Informatique, Université des Sciences et de la Technologie d'Oran, BP 1505, El Mnaouar, Oran, Algeria

Received 20 April 1997; received in revised form 1 November 1997; accepted 20 December 1997

Abstract

The implantation of programs which gives a robot the ability to perform a non-repetitive task (task not completely defined called also unexpected), was hindered by a complex problem: the difficulty met by the classical method in programming (procedural) to formulate a task which the evolution model does not obey to an algorithmic pre-established design. As a part of that the aim of this paper is to propose an analysis approach which lies on a mechanism of factorisation of the complex task. The idea developed consists of subdividing the activity of programming into two steps. A *descriptive step* which allows the formulation of a complex task using a functional approach without integrating any element to the construction of an executing program and a *constructive step* which develops a program using the preceding formulation. This program expresses, more or less explicitly, the *way* of solving different problems posed by the execution of the task at the level of a robot. The aspect of time is introduced as a logical form in the last step for the sequencing of actions while executing a task. © 1998 Elsevier Science Ltd. All rights reserved.

Keywords: Functional description; Unexpected events; Factorisation; Running task; Temporal logic

1. Introduction

Most of the tasks which we propose to *robotise* appear simple and easy for a human operator. This is because the latter usually uses intuitive knowledge or knowledge acquired by experiences. It is not the case of a robot. Here the execution of a task, as simple as it would be, involves the expression of an evolutionary model of the machine in an algorithmic way. So when the robot lacks the intuition allowing it to see when the pre-established algorithm does not describe exactly the needed task, the program construction is extremely complex. A typical example is the mobile robot. The movement of, this latter can be hindered by obstacles, either fixed or mobile, at any moment. The program in which we do not provide the ability to test the obstacles and the procedures corresponding to their processing is totally inoperant.

We must distinguish two classes of the task which we want to *robotize* [1]:

- *the non-repetitive tasks*: these deal with tasks for which it is possible to conceive an organisation of instructions of the executed machine, i.e. the successive steps needed for executing the task.
- *the unexpected tasks*: these tasks are characterised by a very large varying range of execution. It is not easy to establish an algorithm which is able to describe completely the underlying execution process.

In programming, the distinction between the two classes of tasks is very important. It concerns the conception of the program. Therefore, the former concerns the description of a set of instructions that the machine must execute in a defined order (procedural style). The latter expresses the solution to the problem occurring while executing the task i.e. the description of the functionality expected in the program of the task (functional style) [2].

The work presented in this paper concerns the unexpected task. The major point in our problem is to be able to have a set of tools and methods allowing exact expression of the task to be programmed. Furthermore, the different functionality expected in the task's program should be expressed precisely, in an abstract and structured way. We propose an analysis system which incorporates three modules as shown in Fig. 1.

1.1. A functional description of the task

It consists of:

- formulating a task in an abstract manner, i.e. independently of any particular mechanism of implantation, lying on a formal model of that task,
- expressing in terms of that model, the functionality into the program and also the data which are manipulated.

The approach proposed to do the specification of the task is functional. It is a descending method. The user expresses

PII: S0954-1810(98)00001-6

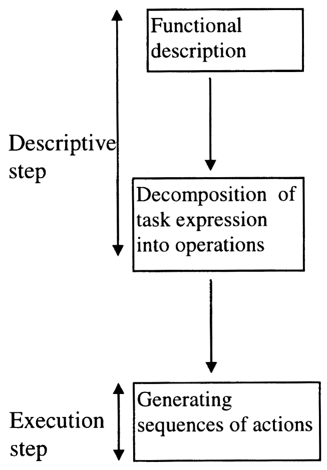

Functional description
Descriptive step
Decomposition of task expression into operations
Execution step
Generating sequences of actions

Fig. 1. A functional factorisation system of task.

first this task in terms of functions that describe the aim to be reached (the result). It aims at expressing what the user wants to do—*his/her function*—without knowing how the program will be implemented. Going from the initial expression, we try after that to have a terminal expression at the end by successive transformation. To such a process a system of *structuration* of the data is associated. Its aim is to distinguish many abstract levels in the description of the data: arriving at a certain step of refinement of the task, a mechanism of abstraction allowing expression of the data needed for doing the abstraction of the other details that are not important.

1.2. Decomposition of task into operations

The terminal tasks are decomposed in terms of *feasible* operations. We say that we do the factorisation of the task. Thus the terminal expression can lead to the derivation of a program which can be interpreted directly by the robot, i.e. it can execute it.

1.3. Generating actions

Actions are generated from the previous transformation of the task into operations. The time notion is used as an Allen model for the sequencing of the actions generated. Such sequencing is the result of the temporal relations.

2. Robotic task description

2.1. Background

In this section we introduce the different levels of the complexity inherent in program construction. Before that, a formal definition of certain basic notions are necessary.

2.1.1. Instruction

It is an entity which allows the description of the action of the machine (robot). Every machine is characterised by a set of finite running instructions.

$I = i_1, i_2, ..., i_n$

2.1.2. Action

The execution of the instruction generates a physical action, changing:

- the state of the robot (the arm position for instance),
- the state of its environment (the object position for instance).

2.1.3. Operation

The operation is an entity to describe the functionality of the robot in the running task. It is characterised by the effect which it is able to produce on that task. It is defined independently of the execution machine.

2.1.4. Robot

In point of view of programming, the robot is an abstract machine called the running machine characterised by:

Intrinsic manner

- a finite set of instructions I, $I = \{i_1, i_2, ..., i_p\}$,
- a finite set of state variables V, $V = \{v_1, v_2, ..., v_m\}$

Extrinsic manner

- a finite set of operations OP, $OP = \{op_1, op_2, ..., op_r\}$
- Therefore, we represent formally the robot by the triplet (I, V, OP).

2.1.5. Running field

It represents the area where the robot evolves, executing the task. This area is defined by a set of objects and it is denoted by A, where $A = \{o_1, o_2, ..., o_n\}$.

2.1.6. Running task

A task is activity supplied by humans and/or the machine

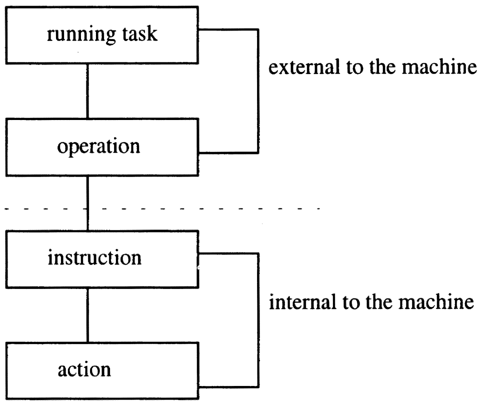


Fig. 2. The programming levels task.

to reach the aim with some conditions. The *running task* term is introduced to formulate a task. It represents a mathematics entity, whose semantic is a function and whose application generates transformations of the running area A.

2.2. A task specification

Many aspects are expected in the expression of the task. From the simple specification of *the aim* to be reached until the prescription of *the instructions* of the machine charged to execute that task, passing by the choice of the operations to be applied. For that, it would be necessary to define *different levels of abstractions* in the programming. For each of those levels it squares *a level expression* (see Fig. 2).

2.2.1. Specification in 'the task level'

In this level, a task is specified in terms of the application of the function like:

assembly (piece: $p1$, piece: $p2$, piece: $p3$)

free (piece: p1) = true
free (piece: p2) = true
in (screw: s2, hole: h2) = true

we are limited to only the aim of the task: the assemblage of the piece p_3 with the initial conditions of the application.

We do the abstraction of the choice of operations to be applied, so we also do the abstraction of the behaviour charged to execute it. We deduce that the specified level is stable. It is independent of any particular implantation mechanism.

2.2.2. Specification in the 'operation level'

In this level we define which operation is chosen to be applied to reach the aim of the task. So we will find:

- the operations which are used,
- the events which enable one or any operations to be started,
- the relations of the preceding binding of those operations, without integrating the expression of *how* the operations are represented or used in the 'execution machine level'.

For example the following sequence:

insert (cylinder: c_1, alesage: a_1)
align (axe: a_1, axe: a_2)
screwed (screw: s_2, hole: h_2)

allows the expression of the previous assembling process in terms of the operations showing the sequencing of different operations which are used (< insert > , < align > , < screwed >).

2.2.3. Specification in the 'machine level'

In this level, we integrate explicitly how the operation used in the previous level can be executed. A task is expressed by a program which is executed directly by some target-machines, thus we can:

- interpret directly the task in terms of instructions which can be executed by the machine,
- represent the evolution of the machine state during the execution of the task.

This is the least stable level specification as it depends closely on the mechanism of the implantation on the running machine.

For example:

MOVEAxis : A0, Speed : 50%.

For the first level we introduce an approach to describe (or specify) a complex task [3].

3. Mechanism of the functional rewriting

We have developed an approach which allows the description of a complex task in a functional and structured way following many abstract levels. That approach brings a descending strategy into operation. It consists first of specifying in terms of functions describing only the aim to be reached (the result): to express 'what' the program 'wants' to do—without taking care of 'how' it will be implemented. That approach encourages the specification of the *task data*, i.e. the properties of the objects in its running space, and *the processing* deal with that task. In other words, the description of the different functionalities of its program.

3.1. Processing specification

Given a running task, the 'processing specification' consists of:

- expressing each complex task with *a canonical function*,
- trying, after that, to define recursively that canonical function in terms of a set of functions simply called auxiliary functions (representing the intermediary tasks), in order to derive a terminal expression, composed only of primitive functions (representing the feasible operations).

The method lies simultaneously on two mechanisms:

1. *the abstract mechanism* leads to a technique of gradual construction distinguishing many abstract levels: an abstract level introduces only the functions needed to describe the functions used in the immediate superior level.
2. *The structuration mechanism* brings a process into operation at each abstract level which refines every *no terminal* task in terms of intermediary tasks. The relationship between tasks and intermediary task are formulated by means of operators called *connectors of the tasks*.

Such a method brings into operation a process of 'growth transformation' for the wording task.

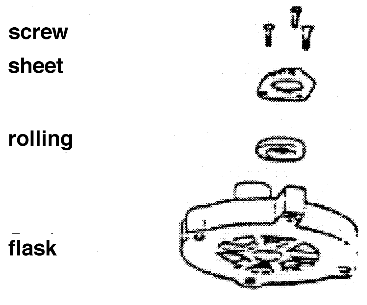


Fig. 3. Assembling sub-assembling$_1$.

3.2. Semantic of the running task

We denote D the domain of the running tasks, the semantic function

$S_T[D] = [D \rightarrow [S_I \rightarrow S_F]]$

means that the semantic of the running task T is a function denoted:

$F : S_I \rightarrow S_F$

where

- S_I is the subset of input states or initial states of T: the states which must be initially satisfied to allow the execution of the task.
- S_F is the subset of the output states or final states: the states which can be reached after the execution of the task T.

3.3. Transformation rules of the task expression

The proposed approach of formulating the transformations of task expression lies on the application of rules called *rewriting rules*. These rules are a simulation of the λ-term substitution model in λ-calculus [4]:

$R[g_1, g_2, ..., g_n)/T_i]T$

'The result expression of changing each occurrence of T_i in T by the expression $(g_1, g_2,..., g_n)$'.

The rewriting rules are defined recursively as follows:

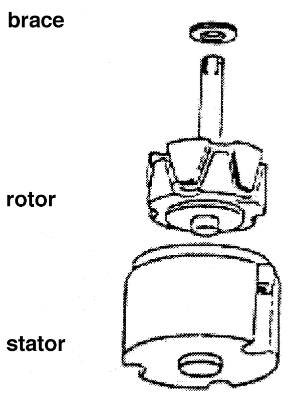


Fig. 4. Assembling sub-assembling$_2$.

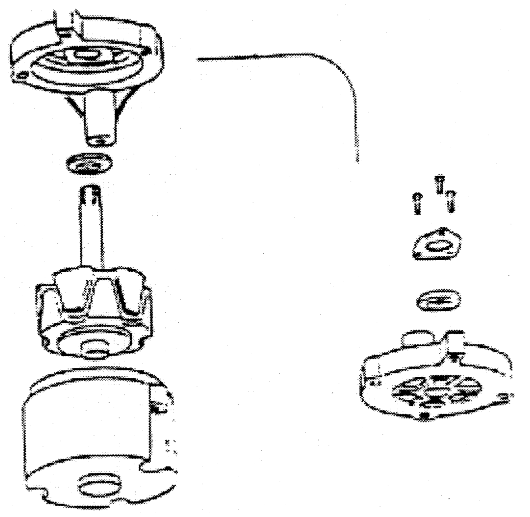


Fig. 5. Assembling sub-assembling$_1$ and sub-assembling$_2$.

1. if $T = T_i$ and $T_i = (g_1, g_2,..., g_n)$ then

   $R[(g_1, g_2, ..., g_n)/T_i]T = (g_1, g_2, ..., g_n)$

2. if $T = [PQ]$ then

   $R[(g_1, g_2, ..., g_n)/T_i][PQ]$

   $= [R(g_1, g_2, ..., g_n)/T_i]P[R(g_1, g_2, ..., g_n)/T_i]Q$

3. if T is a primitive function then

   $R[(g_1, g_2, ..., g_n)T_i]T = T$

3.4. Different forms of the decomposition

3.4.1. Sequential decomposition

Let D be the domain of the running task and O the internal law on D called the product of the task defined by the application $D \times D \rightarrow D$, such that it maps for each couple (T_i, T_j) chosen arbitrarily in $D \times D$, into an element T_k of D. T_k is called the 'sequential product' of T_i and T_j in that order, and defined as follows:

$T_k = T_iOT_j$

T_i and T_j are called factors [5] of T_k.

3.4.1.1. Semantic

The semantic of the sequential product is:

$S[T : d \rightarrow a] = S[T_1 : d \rightarrow a_1]OS$

$\times [T_2 : a_1 \rightarrow a_2]...OS[T_n : a_{n-1} \rightarrow a]$

To illustrate the application for that law we consider the T_1, T_2 tasks shown in Figs. 3 and 4 (T_1: assembly (sub-assembling$_1$); T_2: assembly (sub-assembling$_2$).

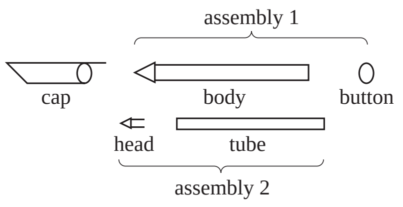


Fig. 6. Assembly of a ball-point pen.

Let T be a task: assemble (piece) (see Fig. 5). The product $T_1 \times T_2$ is the task T and we note $T = T_1 O T_2$. Applying the task T is to apply sequentially T_1 and T_2 in that order.

3.4.2. Parallel decomposition

Let f be a function which we want to decompose in parallel into f_1 and f_2. We note that $f = f_1$ and f_2. Applying f is the same as applying f_1 and f_2, the application order of the functions is not important because their data are totally independent.

3.4.2.1. Semantic

$$[f : d \rightarrow a] = S[f_1 : d_1 \rightarrow a_1], \ldots S[f_n : d_n \rightarrow a],$$

$$\forall i \neq j / (d_i, a_i) \in (d, a),\ (d_j, a_j) \in (d, a)$$

Example

We consider an assemblage of ball-point pens [6] (see Fig. 6).

Let f be a function associated with the following task: to assemble assembly$_1$ and assembly$_2$.

The application of the parallel decomposition supplies two functions f_1 and f_2 as:

Assembly (button, body)
Assembly (head, tube)

The f_1 and f_2 functions can be executed in parallel because their data are totally independent and we note $f = f_1$ and f_2.

3.5. A regular monoide structure

According to the properties of the function product law, the O law is associative. We deduce that the D domain with the O law has a semi-group structure denoted by (D, O).

We suppose that a particular task exists in the D domain denoted by T_e denoting the *idempotence* function. T_e is called the neutral element for the O law as

$$T = TOT_e = T_e OT$$

for every element T of (D, O). The structure (D, O, T_e) is a monoide.

4. Data description

In the formulation of the running task, the data are presented generally as complex objects, where it is necessary to define an adequate formalism allowing the expression of the properties and the operations to be done on the objects. We distinguish two kinds of object: *atomic* and *structured* objects. The underlying description method lies on two mechanisms: data abstraction [7] and data structuration [8].

4.1. Data abstraction

The introduction of such a notion enables manipulation of the expression of the task: complex objects without any reference to implantation structure. So it is possible to express in terms of the abstract type, the representation of the object in the running space: each object is considered as an *instance* of an abstract type.

An object

$$o_i \in \mathrm{OBJ}$$

(set of objects in the running space A) is defined by its type $_i$. We say o_i is an instance of the type $_i$.

For each type defined it corresponds to a sub-set of OBJ called an instance of type and we note that INST($_i$).

The mechanism allowing the relationship between the objects and the types is called the *abstraction mechanism.*

4.2. Data structuration

Data structuration is defined for describing and manipulating the structured objects. Each non-atomic object (decomposable objects) can be decomposed into sub-objects of other types. The structuration mechanism allows the structuration of each non-atomic object into sub-objects. The relationship between the complex object type and the sub-object type is given by the *constructors of type*.

4.3. Object types

We consider two kinds of type:

4.3.1. Atomic type

The atomic type allows the *instanciation* of the atomic objects. The formal model of the atomic object is as follows:

$$OB_{\mathrm{at}} = (\tau,\ N_{\mathrm{op}},\ T_{\mathrm{op}})$$

where

- τ represents a set of attributes ($\tau_1, \ldots, \tau_m$) showing the description of the object properties and its state,
- T_{op} is the set of operations (functions) whose application leads to a transformation of the object,
- N_{op} is the set of operations whose application does not supply a transformation of the object.

Example

Let an object C_1 be declared as follows: C_1: cylinder, where the cylinder type is defined as follows:

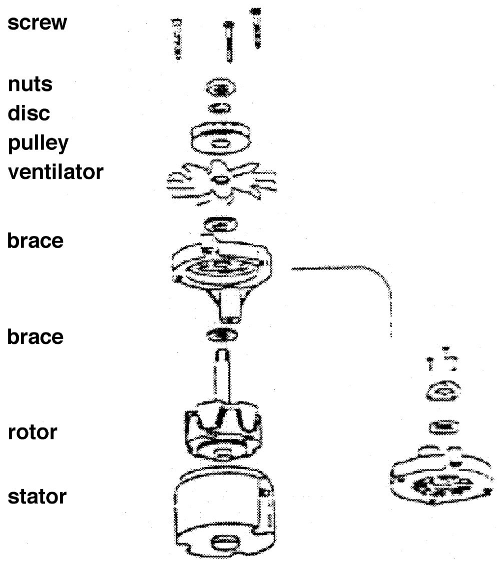


Fig. 7. An alternator structure.

Type cylinder

attribute
row: real
height: real
x, y, z: real
end
signature
take: cylinder → bool;
insert: cylinder × alesage → bool;
put-position: cylinder → real × real × real;
move: real × real × real → real × real × real
end
operation
put-position (cylinder)
take (cylinder)
insert (cylinder, alesage)
move (x, y, z)
end

end

4.3.2. Structured type

They are used for the instanciation of the structured objects. We define the object structured type by using the type constructors [9]. We have two kinds of constructors:

4.3.2.1. 'Disjointe union' constructor type

Let $_1$,..., $_n$ be a set of types

ξ_u is the type denoted $\xi_u = \text{Union}(\xi_1..., \xi_n)$,where Union represents the disjoint union type constructor whose semantic is defined by the following equation:

$$\text{INST}(\xi_u) = \text{INST}(\xi_1) \oplus \text{INST}(\xi_2)...\text{INST}(\xi_n)$$

where the $\oplus$ symbol is the operator of the disjoint union of sets.

Example

Type sort_of_fixing

union (rivet, screw)

the object type sort_of_fixing is composed by rivet or screw. We define the object one_fixing by its type sort_of_fixing:

one_fixing: sort_of_fixing

4.3.2.2. 'Cartesian product' constructor type

Let $_1$..., $_n$ be a set of types.

ξ_p is the type denoted $\xi_p = \text{Prod}(\xi_1..., \xi_n)$, where Prod represents a constructor of the cartesian product type whose semantic is given by the following equation:

$$\text{INST}(\xi_p) = \text{INST}(\xi_1) \times \text{INST}(\xi_2)... \times \text{INST}(\xi_n)$$

Example
For an alternator structure see Fig. 7.

Type alternator

Prod (screw, screw, screw, nuts, disc, pulley, ventilator, brace, screw, screw, screw, sheet, rolling, flask, brace, rotor, stator).

4.4. Object constructors

We consider two kinds of object constructors: the union and the cartesian product object constructors.

4.4.1. Object union constructor

Let O_u be an object of the type Union (ξ_1, ξ_2..., ξ_n) which is a set of the sub-objects defined as follows:

$$O_u = \boldsymbol{U}(O_1, O_2..., O_p)$$

where

- an object $O_i \in \text{INST}(\xi_i)$
- U represents an object constructor of the type union.

Example
One_fixing = $\mathbf{U}(r_1, v_1)$, $r_1 \in$ INST(rivet), $v_1 \in$ INST(screw).

4.4.2. Cartesian product object constructor

An object O_p of the type Prod(ξ_1, ξ_2..., ξ_n) is a set of sub-objects defined as follows:

$$O_p = \boldsymbol{P}(O_1, O_2..., O_n)$$

where

- an object $O_i \in \text{INST}(_i)$
- P represents an object constructor of cartesian product type.

Example
We consider the object alternator defined previously. It is

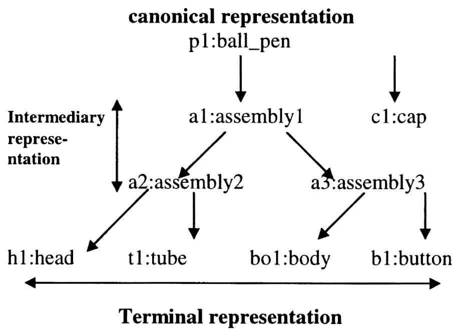


Fig. 8. Structure of an object of the ball-point pen type.

built by using the cartesian product object constructor:

$$\text{an} - \text{alternator} = \boldsymbol{P}(s_1,\ s_2,\ s_3,\ n,\ d,\ p,\ v,\ b_1,\ s_4,\ s_5,\ s_6,\ sh,\ rl,\ f,\ b_2,\ rt,\ st)$$

4.4.3. Formal model of structured objects

For each structured object of the running space A is associated a structured type ξ_c. This type is defined by using the constructors of the type presented previously.

Formally a structured object is a couple

$$OB_s = (C,\ S_{ob})$$

where C represents a constructor of objects and S_{ob} is a subset of sub-objects constituting the structured object.

4.5. Different representations of structured objects

Thanks to mechanisms of abstraction and structuration, it is possible:

- to represent an object following the processing we wish to do on it by doing the abstraction of the others.
- to extract the relevant information for doing some processing on an object.

Consequently, there exist many representations for a structured object:

- canonical representation,
- intermediary representation,
- terminal representation.

4.5.1. Canonical representation

Let O be an object from the OBJ domain. We call *canonical representation* [10] of O at a given abstract level, the n-uplet denoted $R_c(O)$ and defined as:

$$R_c(O) = (O_1, O_2..., O_n)_{co} \qquad O_i \in \text{OBJ}$$

where the operator $()_{co}$ is the object constructor P and U, and O_1, O_2..., O_n are the sub-objects of O.

The aim of the canonical representation is to lead to a complete description of structured object (non-atomic) in terms of *an aggregate of object*.

4.5.2. Intermediary representation

It often happens that the canonical representation does not allow one to distinguish the pertinent properties of the property set which the object has when we apply a particular processing on that object. So, the suggested way is to divide at each abstract level the canonical representation $R_c(O)$ into a set of sub-representation, denoted:

$$R_I = r_1(o),\ r_2(O)...,r_n(O)$$

defined as:

(i) $r_p(O) =< O_{p1},\ O_{p2}...,O_{pn} >$

(ii) $\forall i \in [1,n]O_{pi} \in \{O_1...,O_n\}$

The interest of the abstract mechanism is double:

- to give a basic representation, but which is efficient and displays for each type of intermediary processing, the most interesting properties of manipulated objects by doing the abstraction of the other,
- to create a relationship of dependency that assures a good match between the data structuration and the refinement of processing.

4.5.3. Terminal representation

A representation is called *terminal* if at the end of decomposition, each structured object is substituted by atomic objects,

$$R_t(O) = O_i/O_i \in OB_a$$

where OB_a is a set of atomic objects.

Example

The decomposition of the ball-point pen object is represented in Fig. 8.

5. Decomposition of task into operations

The aim is to try to have in the terminal expression only the feasible *factors* i.e. the factors which are by hypothesis interpreted in terms of a set of feasible operations for a certain target machine.

5.1. Definition of feasible operation

An operation is called feasible for a robot if it can be expressed in terms of a finite instruction sequence that can be executed by that robot.

5.2. Notion of factorisation

The notions of rewriting rules have been introduced allowing the transformation of the running task expression. We will see afterwards the important point which is *factorisation* of such an expression: it deals with formulation of a process of successive transformations of that task expression.

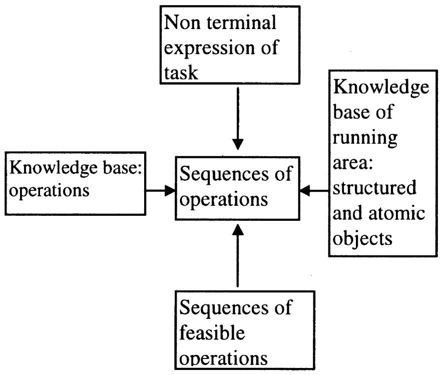

Non terminal expression of task
Knowledge base of running area: structured and atomic objects
Knowledge base: operations
Sequences of operations
Sequences of feasible operations

Fig. 9. A transformation of task into operations.

Let T be a task and (T_i) $i = 1, 2...n$, a queue task with $T = T_n$. T_n is called canonical expression of T.

The W set defined as:

$$W = \{w_i/T_{i+1} = w_i \text{ and } T_i \text{ or } T_{i+1} = w_i O T_i \; n < i < 0\}$$

is called a factor set of the T task.

The multi set defined on W is called the *factorisation* of the T task.

The expression

$$w_{n-1} O w_{n-2} O ... w_1 O w_0$$

or

$$w_{n-1} \text{ and } w_{n-2} \text{ and } ...w_1 \text{ and } w_0$$

is called a factorisation form of the T task and we note

$$T = w_{n-1} \text{ and } w_{n-2} \text{ and } ...w_1 \text{ and } w_0$$

or

$$T = w_{n-1} O w_{n-2} O ... w_1 O w_0$$

5.3. Terminal task

Let T_k be a task of D domain; T_k is called irreducible or terminal, if it can be factoring.

We note D_t the sub-domain of D for the terminal task. Let T be a task; a factorisation of T is called terminal if each factor appearing in the factorisation form of T belongs to sub-domain D_t.

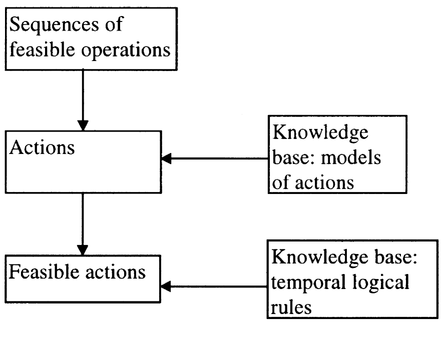

Sequences of feasible operations
Actions
Knowledge base: models of actions
Feasible actions
Knowledge base: temporal logical rules

Fig. 10. Generation of feasible actions.

5.4. Feasible factorisation

Let T be a task of D domain and $w_{n-1} O w_{n-2} O ... w_1 O w_0$ a factorisation of the T expression.

We say that we deal with a *feasible factorisation* if for each factor w_i for $i = 0, 1..., n - 1$ that appears in the factorisation form, we can formulate an interpretation in terms of an ordered finite multi set of feasible operations, i.e.

$$\forall i \in [0, n-1]/(C(w_i), \; M_i)$$

where $C(w_i)$ is the operational context of w_i and M_i is the running machine charged for interpreting the operations defined in $C(w_i)$.

$$T = w_{n-1} O w_{n-2} O ... w_1 O w_0 \forall i \in [0, n-1] \text{ and } M = M_i.$$

We say that T is feasible for the machine M.

5.5. Transformation of the task expression into operations

The generation of the operations is in relationship with:

- the environment of the task, which is defined as a set of abstract objects, with possible representations for those abstract objects,
- the set of basic operators for transforming those representations,
- a class of predicates that are able to be evaluated on those data structures.

A knowledge base contains data structure, describing the characteristics of those structured and abstract objects.

Using on the one hand the information contained in that base and in the other hand a library of the operations [11], given a running machine (robot), the expression of the task will be translated into sequences of feasible operations (see Fig. 9).

6. Generation of feasible actions

We want to express a task with a program directly executed for a target machine lying on an organisation of target machine instructions [12]. It involves that we can:

- interpret directly the task in terms of instructions executed for the target machine. For example in terms of moving as: MOVE axis: A0, speed: 50,
- represent the evolution of the immediate state of the machine during the task execution.

6.1. Temporal factorisation of actions

Each operation generated by the decomposition will in its turn be decomposed into a set of ordered actions, by consulting a knowledge base which contains some characteristics of feasible instructions of the target machine. The sequence of actions is generated [13] with the help of the temporal rules base (see Fig. 10).

The notion of time and events are introduced in the

relation	inverse	picture
I_1 before I_2(<)	>	I1 I2
I_1 meets I_2 (m)	mi	I1 I2
I_1 starts I_2 (s)	s^{-1}	I1 I2
I_1 over I_2 (o)	o^{-1}	I1 I2
I_1 during I_2 (d)	d^{-1}	I1 I2
I_1 equal I_2 (=)	=	I1 I2
I_1 finish I_2 (f)	f^{-1}	I2 I1

Fig. 11. The 13 Allen's relations.

generation of actions. The ordering of generated action is done by using some temporal rules [15]. The Allen model is used as a logical tool for temporal reasoning on the actions [14, 15].

Such logic is based on temporal interval rather than time points. It is typed first order predicate calculus. There is a basic set of primitive relations that can hold between temporal intervals. Each of these is represented by a predicate in the logic: the sequencing of the actions is determined by the temporal relationships between the different time interval.

Example

Let A_1, A_2 be two actions and I_1, I_2 two corresponding time intervals. We give the different relations that exist between intervals [16] (see Fig. 11).

There are small numbers of predicates. One of the most important is HOLDS that asserts property holds (i.e. true) during the time interval. Thus HOLDS(p, t) is true if, and only if, property p holds during t.

Allen defined another predicate OCCUR which takes an event and a time interval and is true only if the event happened over the time interval and there are no subintervals of t over which the event happened.

6.2. Example of action generated from the operation

Let us consider the example of the assembly of the ball-point pen more precisely, the assembly$_2$: the operation screw H1 of the type head into T1 of the type tube is decomposed into three actions.

To assemble H1 and T1:

- action take H1
- action take T1
- action introduce H1 in T1

(1) Take H1 (action A_1)

The preconditions of that actions are

- free H1: H1_f
- free pliers: P1_f

The effects

- H1 into pliers: H1_P1

Description of the action Take H1:

IF OCCUR (Take H1, t_1) THEN & HOLDS(H1_f, t_2) & Meets(t_2, t_1) & HOLDS(P1_f, t_3) & Meets(t_1, t_3) & HOLDS(H1_P1, t_4) & Meets(t_1, t_4)

(2) Take T1 (action A_2)

The preconditions

- free T1: T1_f
- free pliers: P2_f

The effects

- T1 into pliers: T1_P2

Description of the action Take T1:

IF OCCUR(Take T1, t'_1) THEN & HOLDS(T1_f, t'_2) & Meets(t'_1, t'$_2$) & HOLDS(P2_f, t'_3) & Meets(t'_1, t'_3) & HOLDS(t1_P2, t'_4) & Meets(t'_1, t'_4)

(3) Introduce H1 into T1 (action A_3)

The preconditions

- H1 into pliers: H1_P1
- T1 into pliers T1_P2

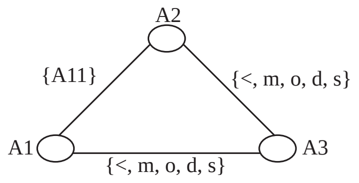
The effects

- H1 into T1: H1_T1

IF OCCUR(Introduce_H1_P1, t''_1) THEN & HOLDS(H1_P1, t''_2) & Finish(t$''_1$, t''_2) & HOLDS(T1_P2, t''_3) & Finish(t''_1, t''_3) & HOLDS(H1_T1, t''_3) & Meets((t''_1, t''_4)

Some relations between time intervals into each actions are deduced. For instance, for the action Take H1, the following are the relations between time intervals t_1, t_2, t_3, t_4 : $t_1\ m\ t_2$, $t_1\ m\ t_3$, $t_2\ f\ t_3$, $t_2 = t_3$, $t_2\ fi\ t_3$, $t_4\ s^{-1}\ t_3$, $t_4\ o^{-1}\ t_3$, $t_{4c}\ d^{-1}\ t_3$, $t_1\ m\ t_4$

So Relations between actions i.e. the sequencing, are as follows:



7. Conclusion

In the area of programming running tasks, the conception of the robots able to accomplish more and more complex tasks must now respond to the will and/or the necessity to build programs leading to describing unexpected tasks, i.e. non-repetitive tasks, where the scenario of execution does not necessarily obey an algorithmic pre-established design. Thus the task of the user is a priori hard. The user is unlikely to succeed unless by recourse to a rigorous approach allowing an accurate and structured description from the non-explicit initial expression, (abstract levels), from which it is possible later on to derive an algorithmic shape.

The interest of the approach developed in this paper is to allow, from the preliminary step (called also descriptive), to break off the strict constraints of the algorithmic check. It deals with supplying a static description i.e. non-algorithmic, where, following a structured method, we specify the different functionalities expected in the program that describes the execution of the considered task by the robot. Such a description excludes any element of resolution of the problem. The user tries to formulate the task to compute in a structured and abstract way in terms of a set of functions, expressing each more precise intermediary task. These tasks in their turn can be refined in intermediary tasks... and recursively to derive a formulation of the global task in terms of the terminal task expressed in terms of operations. We notice the absence of any algorithmic shape in the wording in this way. For instance, the ordering of different terminal tasks is pushed away in the final step called the executing step. In this step the order of execution of these tasks is deduced from logical–temporal relations which are closely dependent on the context of execution of the global task.

The work presented in this paper does not require to cover all the aspects involved from the wording of the task. Nevertheless, the reflections which are done contribute to enrich the underlying ideas.

Acknowledgements

The author is particularly grateful to K. Ouriachi of the Département d'Informatique at the University of Pau, and to Z. Mekakkia, S. Selka, A. Chouarfia and B. Messabih of the Institut d'Informatique at the USTO University. The author also thanks the anonymous referees for many useful suggestions, and their constructive criticism.

References

[1] Kalafate O. Planification des tâches opératoires robotiques basée sur le modèle d'acteur. PhD thesis, Université de Valenciennes, France, 1990.
[2] Backus B. Can be liberated from the Von-Neumann style? A functional style and its algebra of programs. ACM 1978;21:613–640.
[3] Benbernou S, Ouriachi K. A complex task modelling. Proc IFAC Symposium. Oxford: Pergamon Press, 1991.
[4] Barendregt HP. The lambda-calculus. Its syntax and semantics. Studies in Logic and the Foundations of Mathematics, Vol. 103, 1985.
[5] Ouriachi K. Contribution à la spécification formelle des tâches opératoires: outils de base en vue d'une approche fonctionnelle basée, sur l'agrégation des données. PhD thesis, University of Valenciennes, France, 1991.
[6] Fazio D, Whitney DE. Simplified generating mechanical assembly sequences. IEEE Journal of Robotics and Automation 1987;3(6).
[7] Musser DR. Abstract data type specification in the AFFIRM. IEEE Transaction on Software Engineering 1980;6:1.
[8] Woods WA. Important issues in knowledge representation. Proc IEEE 1986.
[9] Benbernou S. Outils de base en vue d'une approche fonctionnelle de programmation des tâches opératoires: application à la construction de programme des tâches d'assemblage. PhD thesis, University of Valenciennes, France, 1991.
[10] Abramsky S, Hankin C. An introduction to abstract interpretation. Abstract interpretation of declarative language. Ellis Horwood, 1987.
[11] Kalafate O, Ouriachi K, Bourton M. Assembly tasks planning with categories. 10th Int. Conf. on Assembly Automation, ICAA, Kanazawa, Japan 1989.
[12] Hustung K, Wee WG. A Knowledge-based planning system for mechanical assembly using robot. IEEE Transactions 1988.
[13] Kiam Seow, Devanathan R. Temporal logic programming for assembly sequence planning. Artificial Intelligence in Engineering 1994;8:259.
[14] Rutten E, Marcé L. An imperative language for task-level planning: definition in temporal logic. Artificial Intelligence in Engineering 1994;8:235.
[15] Allen J. Maintaining knowledge about temporal intervals. Communication of the ACM 1983;26:832–843.
[16] Allen J. Towards a general theory of action and time. Artificial Intelligence Journal 1984;32(2):123.