

***Program Supervision: from Knowledge Modeling to
Dedicated Engines***

Sabine Moisan Régis Vincent Monique Thonnat

N° 3324

Décembre 1997

THÈME 3



*R*apport
de recherche

Program Supervision: from Knowledge Modeling to Dedicated Engines

Sabine Moisan * Régis Vincent[†] Monique Thonnat[‡]

Thème 3 —Interaction homme-machine,
images, données, connaissances
Projet ORION

Rapport de recherche n° 3324 —Décembre 1997 —34 pages

Abstract: In this report we present knowledge-based techniques for automating the use of a modular set of programs (or program supervision in short). Our goal is to provide tools which are both general and flexible. First we describe how program supervision knowledge can be modeled and we propose a knowledge description language, YAKL, to easily express this knowledge. Then we present two dedicated program supervision engines which are extensions of the OCAPI program supervision engine. The PEGASE engine provides mechanisms for managing sophisticated repair strategies. The second one PHENIX is a first attempt to introduce more dynamics in the planning phase of PEGASE. Examples are shown through an image processing application, the morphological description of galaxies.

Key-words: knowledge-based systems, program supervision, planning, image processing, software reuse

* Email : Sabine.Moisan@sophia.inria.fr

[†] Email :Regis.Vincent@sophia.inria.fr

[‡] Email :Monique.Thonnat@sophia.inria.fr

De la modélisation du pilotage de programmes vers des moteurs dédiés.

Résumé : Dans ce rapport nous présentons des techniques d'intelligence artificielle pour automatiser l'utilisation d'un ensemble modulaire de programmes. Ces techniques seront appelées pilotage de programmes dans la suite. Le but est fournir des outils à la fois généraux et souples. Premièrement nous décrivons comment modéliser les connaissances en pilotage de programmes et nous proposons le langage de description YAKL pour exprimer facilement ces connaissances. Nous présentons ensuite deux moteurs dédiés au pilotage de programmes qui sont des extensions du moteur d'OCAPI. Le moteur PEGASE fournit des mécanismes pour gérer des stratégies de réparation sophistiquées. Le second, PHENIX, constitue une première tentative pour introduire plus de dynamisme dans la phase de planification de PEGASE. Des exemples de pilotage de programmes sont montrés à travers une application de traitement d'images pour la description morphologique de galaxies.

Mots-clés : systèmes à base de connaissances, pilotage de programmes, planification, traitement d'images, réutilisation de logiciels

1 Introduction

In this paper we present knowledge-based techniques for managing the use of complex software. More precisely we are concerned with **program supervision**.

In program supervision the complex software to manage is a modular set of programs. The role of program supervision is to select the adequate programs in an existing library, to schedule them, to compute the values of their parameters, to run the programs and eventually to control their execution, for a particular context.

So, program supervision consists in different phases, that can be completely or only partly automated: *planning* and *execution* of programs, *evaluation* of the results, and *repair* (see figure 1). The plans generated by the planning phase are composed of operators which are programs.

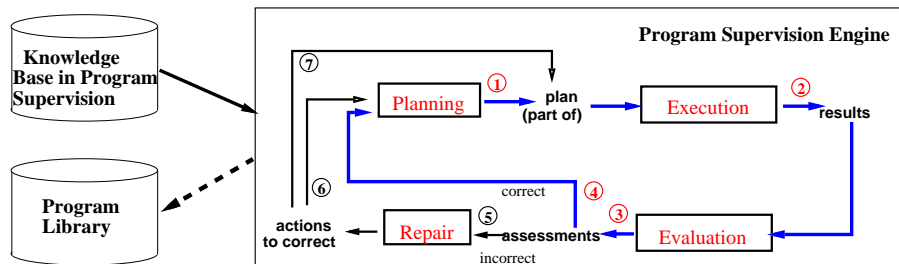


Figure 1: *Planning* first builds a plan, or a part of plan (1), *execution* executes this (part of) plan, and produces results (2), these results are given to *evaluation*, which returns assessments (3). If the assessments are correct (4), the planning process can go on. If failures have been detected (5), from the failures and the (part of) plan that has been executed, *repair* decides which correcting actions are appropriate. This will either run again the planner (6) or the executor (7).

Every end-user could not have a deep understanding of the program semantics and syntaxes. On the other hand, programs implement more and more complex functionalities and their use is more and more subtle. If it is too demanding for an end-user to catch the complexity of new programs, they will stay in the laboratories, and never be widely applied. The use of existing libraries of programs has become a critical resource in many disciplines. Many different libraries of programs have been developed in domains like signal processing, image processing, and scientific computing. These libraries consist of a large number of programs, written

by specialists in a particular domain, and applied by non-specialist in this domain. Recently artificial intelligence techniques have been used to help a non-specialist to apply the programs in different working environments. Indeed, a knowledge-based system may take in charge the management of the library use, freeing the user of doing it manually. This aid can range from an advisory guide up to fully automatic program monitoring systems.

Program supervision is a recent research domain with an increasing number of work coming from many applicative and technical domains [38]. These research activities are often motivated by a particular application domain (as image processing, signal processing or scientific computing).

On the contrary to knowledge-based systems for image interpretation which have been studied for about 15 years, knowledge-based systems for image processing program supervision are recent. In Japan, a lot of teams, belonging both to the research and to the industrial sectors, have devoted an important effort on this problem. [39], [31], (see the review made by Matsuyama in 89 [26]). In the United States early work has been done by Johnston [22] and by Bailey [1]. In Europe work has been developed either as research general tools (see OCAPI [10] and [11]) or as industrial tools for a particular application (see the VIDIMUS Esprit project [6] and [3]). In the VIDIMUS project the aim was to develop a vision system environment for industrial inspection applications. The result was a knowledge-based system (VSDE) that can be used to first specify an inspection problem, and to automatically configure a vision based system to solve the problem.

The problem of program supervision is even more recent in signal processing and automatic control. In signal processing one can mention the IPUS system [29] which has been applied to sound recognition. In automatic control, expert systems have been developed for a specific task (identification); one can mention the following systems: IHS [24], Espion [19], SEXI [18]. The work described in [13, 32] is interesting in that although it is currently restricted to the problem of system identification, it aims at creating a wider platform for the development of signal processing applications. A brief review is presented in [34].

In scientific computing, a lot of work concerns problem solving environments (like Mathematica ou Matlab) [17, 20]. These environments usually provide state-of-the-art methods and interactive color graphics. On contrary few work is really related to program supervision. Among them we can cite the following systems:

FALCON [14], SCAI ([30]), and SCARP ([46]). The effort is often stressed on the interaction with the final user and not on the full automation of the problem solving.

In software engineering the objective of software reuse is very close to some program supervision sub-problems. More precisely the planning subproblems of selection and scheduling are very important for program reuse and for software development method reuse [2, 23]. A brief review of these problems can be shown in [40]. Software engineering can be viewed w.r.t. program supervision either as an application domain or as technique provider.

In artificial intelligence program supervision is not a research domain per se, although researches are carried out on a connected notion: abstract task. For instance, one can cite the following work [5, 4] on conceptual modeling, [44] on tasks, methods and mechanisms, and [8] on task structures. These research axes have methodological purposes which can be very useful for knowledge acquisition in program supervision.

Our goal is to provide models and tools which are both **general** and **flexible**. As program supervision is a general problem arising in various application domains we are interested in providing both knowledge models and software tools which are independant of any particular application and of any library of programs. Another important feature for program supervision is the flexibility. The lack of certain type of knowledge has to be compensated by powerful control mechanisms, like sophisticated repair mechanisms.

In this paper, first we briefly present a typical program supervision problem, the automation of galaxy image processing; this application is used both to illustrate knowledge examples and program supervision engine behavior. Then in section 3 program supervision knowledge modeling is studied in detail and a knowledge description language named YAKL is described. In section 4, we propose two dedicated program supervision engines compatible with the previous knowledge modeling. Finally in section 5, we present a software development platform used both to speed-up the development of new program supervision engines and to share common tools enabling easy comparison of different models and behaviors.

2 A program supervision application

In this section we introduce a typical program supervision problem in the domain of image processing. The role of the modular set of image processing programs is to analyse astronomical images containing a galaxy. The long term goal is to classify the galaxy contained in the image as astronomical experts do [35, 36]. Furthermore, because astronomical observing systems provide a great amount of high quality data, it is necessary to automate their processing. The role of the image processing phase is to describe the morphology of the galaxy in terms of pertinent numerical parameters. In the current version of the system 45 numerical parameters are computed : 3 global morphological parameters (size, orientation, excentricity) of the galaxy, 2 global photometrical parameters (measuring the errors made by approximating the projected profile with a linear and a $r^{\frac{1}{4}}$ model) and 8 local parameters describing the morphology of 5 different regions of the galaxy (orientation, perimeter, area, coordinates of the center of gravity, length on the major axis and on the minor axis and distance to an elliptical shape). So for each image, the goal of the image processing is the same : detection of the galaxy and computing of these global and local parameters (see figure 2).

Although, because of the great variability in the images, which are obtained with different observation conditions (photographic plates or CCD cameras, high or low resolution telescopes, long or short exposure time, presence in the image of other objects (stars), distance of the observed galaxy, *etc.*) both the sequences of procedures, and the values of their parameters need to be adapted. Some examples of the variety of images we have to deal with are shown in figure 3. For instance, images a) and c) correspond to noisy images containing a low resolution irregular and elliptical galaxy. Images b) and d) are good quality images containing high resolution galaxies; image d) is disturbed by the presence of numerous stars in the field of view.

The solution we propose to solve this problem is to explicit, in a knowledge base, the criteria we use to take into account these variations and to connect this knowledge base with a library of image processing programs [36]. A first knowledge-based program supervision system [37] has been done with OCAPI [11].

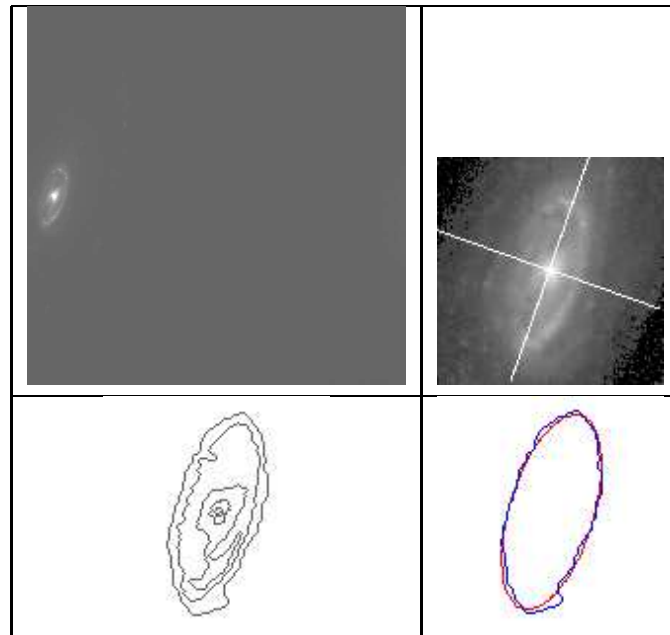
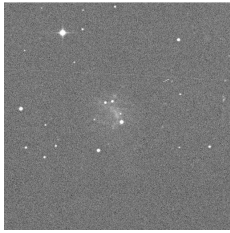


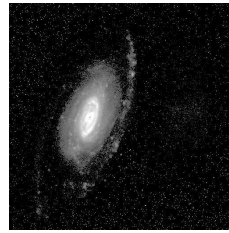
Figure 2: The different steps of galaxy image processing: top left: the original image (galaxy NGC7531); top right: the isolated galaxy with its main axes; middle left: five iso-intensity contours; middle right: approximation of the external contour with an ellipse.

We can distinguish several typical decompositions of image processing into sequential, parallel or optional sub-steps. The global processing named morphological-description is described by a decomposition into five sequential sub-steps :

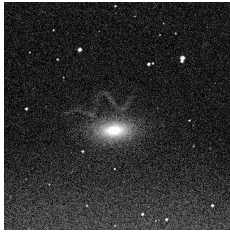
- creation and initialization of the file containing the numerical parameters,
- extraction of the object of interest (the galaxy itself)
- computing of 5 parameters describing globally the galaxy
- building five iso-intensity contours corresponding to different regions of the galaxy



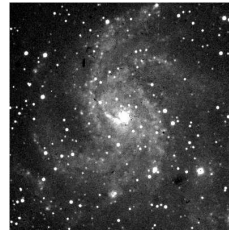
a)Galaxy NGC4523



b)Galaxy NGC7531



c)Galaxy NGC4473



d)Galaxy NGC6946

Figure 3: Examples of the image variety: a) a noisy image containing in its center a faint irregular galaxy b) a good quality image containing a high resolution uncentered spiral galaxy c) a noisy image containing in its center a low resolution elliptical galaxy d) a good quality image containing a large high resolution galaxy superimposed with numerous stars

- for each built contour, computing of a set of 9 parameters describing its area, perimeter, distance between the contour and an ellipse...

Some of these steps are complex and are decomposed into sub-steps. The most sensitive and complex part is the extraction of the object of interest; the extraction of the object of interest is a very important step on which depends the quality of the final result. The system must be able to take into account several conditions and must be able to evaluate the intermediate results of the processing and if necessary to adjust the treatment. The extraction itself consists of three main steps which are : the localization of the center of the galaxy, the effective isolation of the galaxy and noise removal. These steps are themselves decomposed into sub-steps. For instance, depending on the context of utilization the image has to be processed in two possible ways : if the galaxy is centered in the image, a sub-image is extracted

around the center of the image, if we have no a priori information on the position of the galaxy, the system looks for the most important object present in the image (biggest extended source). Then it is needed to check if the system has actually detected the right object (a star may be detected). This can be done by verifying that the most important detected object has a size larger than the maximal size of a star. The complete treatment uses 38 programs.

3 Knowledge modeling

3.1 Analysis of the reuse problem

When analysing the activity of (re)using a number of complex programs for an applicative purpose, we have identified the difficulties that arise from the processing complexity, independently of any application. Here follows the modeling efforts an end-user must do in order to have his/her data correctly processed, and to use efficiently the set of programs:

1. Build a model of programs, i.e. for each program:
 - Understand its purpose and behavior.
 - Remember the number and types of its arguments, e.g. the type of data it accepts as inputs.
 - Know its precise calling syntax, together with its usual parameter values.
2. Model program combinations:
 - Figure out which programs can be combined together and how - e.g. what are the data flows between programs.
 - Remember useful program combinations, for typical processing goals.
 - Know how to choose among multiple alternatives
3. Model repair strategies: if, at any moment during processing, the current results are not as good as expected
 - Infer which previously executed program is faulty.

- Decide whether to rerun with new parameter values - and how to compute the values - or to replace it by another program - and by which one.

3.2 Proposed PS model

From this analysis of the task of a human-being processing data, using a set of programs, we have derived knowledge models related to this activity, that we call program supervision.

First programs have to be modeled. A program model contains its complete calling syntax, descriptions of its input and output arguments (e.g. name, type, usual parameter values, etc.) and its conditions of use.

Typical program combinations, if they are known, may also be profitably modeled by describing their decomposition into more concrete programs, at different levels of abstraction, either by specialization (alternatives) or composition (sequences, parallel, loops, etc.).

Data model describes precisely the details of data types, their sub-parts, etc. Data play an important role in program supervision because many decisions are based on the information that they provide. This is particularly true if processing is data-driven as in image processing. Some domain objects may also be used during reasoning, those objects are highly dependent on the application domain and can be modeled by the expert.

Finally various criteria play an important role in the model to choose between different alternatives, to tune program execution, or to diagnose, and to repair a bad execution. Each criterium is related to a particular sub-part of program supervision: choice criteria are connected to the planning phase, tuning criteria to the execution phase, etc.

3.3 YAKL: a Knowledge Description Language

We propose a language, named YAKL, that provides a “model-view” of the expert knowledge about a set of programs, independently of any application domain, of any program library, or of the implementation language of the knowledge-based system (in our case Lisp or C++). The YAKL language is used both as a common

storage format for knowledge bases and as a human readable format for writing and consulting knowledge bases.

Since program supervision consists in arranging relevant operators to achieve a processing goal, given some initial data, a knowledge base for program supervision should describe the goals, the operators with their arguments, the data, the conditions under which operators are applicable, the possible relations between them, etc.

YAKL allows the expression of all these different types of knowledge on program supervision. The language proposes two types of declarative descriptions: structural frame-based and rule-oriented. Structural descriptions are used for goals, operators, data and arguments, while rules are used for criteria expression.

3.3.1 Structural descriptions

Goals are described by structures that express an abstract functionality to achieve, together with constraints on the expected final state.

Operators describe how to achieve a goal. Individual programs are modeled as **Terminal Operators** and known program combinations by **Complex Operators**. Different operators (terminal or complex) can be applied to achieve one single goal.

Operator representation includes:

- a symbol describing the abstract functionality of the operator;
- an optional list of symbols describing the characteristics or typicalities of an operator;
- information on arguments (input data, input parameters, output data) including their names and types;
- pre- and post- conditions that state when the operator is applicable and what should hold after the application of the operator.

In addition to this information, a terminal operator contains a description of the calling syntax. The following example shows the terminal operator `o-muls`. Only the structural part is shown, criteria are detailed later (see section 3.3.2). YAKL keywords are in bold face:

```

Terminal Operator {
    name : o-muls
    comment : "Thresholding (inrimage)"
Functionality : thresholding
Input Data
    INRI name : input-image
    comment : "image to threshold"
Input Parameters
    Float name : threshold
    default : 1
Output Data
    INRI name : output-image
    comment : "thresholded image"
    I-O Relations :
        output-image.path := input-image.path,
        output-image.basename := input-image.get-simplename,
        output-image.extension := ".muls",
        output-image.x-size := input-image.x-size,
        output-image.y-size := input-image.y-size,
        output-image.x-sample-step := input-image.x-sample-step,
        output-image.y-sample-step := input-image.y-sample-step
Preconditions
    valid input-image
Postconditions
    valid output-image
Criteria
    ....
Call
    language : shell
    syntax : cd input-image.path ";" muls -vs threshold output-image
    program-name : muls
    type : real
}

```

The way a **Complex Operator** has to be refined is expressed in a body containing a description of its decomposition into sub-elements (type of decomposition and name of sub-elements) and data flow information between father and sons (named

distribution) and between sons (named flow). The allowed types of decompositions are alternative, sequential, parallel and optional. These decompositions at different levels of abstraction must end on terminal operators. The same terminal or complex operator may appear in different decompositions.

The following example shows the complex operator o-detection. Its body is a sequential decomposition into 5 substeps among which are the four terminal operators o-thres-muls, o-muls, o-morphlis and o-boite, and one complex operator construct-contour-chain.

Complex Operator {

name : o-detection

comment : "Detection of the zone of the galaxy"

Functionality : galaxy-detection

Input Data

INRI **name :** input-image

Float **name :** sm

Float **name :** saire1

Float **name :** saire2

Output Data

INRI **name :** output-image

Integer **name :** ix

Integer **name :** iy

Integer **name :** x

Integer **name :** y

Integer **name :** size

Preconditions

valid input-image

Postconditions

valid output-image,

valid ix,

valid iy,

.....

Criteria

....

....

Body

o-thres-muls - o-muls - o-morphlis - construct-contour-chain - o-boite ;

Distribution

```

o-detection.sm / o-thres-muls.sm
o-detection.saire1 / o-thres-muls.saire1
o-detection.saire2 / o-thres-muls.saire2
o-detection.input-image / o-muls.input-image
o-detection.output-image / o-boite.output-image
o-detection.ix / o-boite.ix
o-detection.iy / o-boite.iy
o-detection.x / o-boite.x
o-detection.y / o-boite.y
o-detection.size / o-boite.size

```

Flow

```

construct-contour-chain.fchaine / o-boite.input-image
o-morphlis.output-image / construct-contour-chain.input-image
o-thres-muls.threshold / o-muls.threshold
o-muls.output-image / o-morphlis.input-image
}

```

Structured types can also be described both for arguments of operators and domain objects that depend on the application. YAKL provides a standard frame-based representation for types with a hierarchical organization. Both data and domain objects types descriptions can include methods associated to them, e.g. display methods. Arguments are associated with operators (terminal or complex) and goals. YAKL distinguishes two classes of arguments, data and parameters. Data arguments have fixed values which are set or computed, while parameters are tunable.

Example:

```

Argument Type {
  name : INRI
  comment : "Definition of INRIImage type"
Subtype Of : Image
Attributes
  Integer name : x-sample-step
    comment : "thresholding coefficient"
    default : 1
  Integer name : y-sample-step
    comment : "thresholding coefficient"
    default : 1
}

```

3.3.2 Criteria Rules

Different criteria are used to decide how to choose among methods, how to initialize input arguments, how to evaluate results, how to adjust the processing with the determination of new input values for programs or selection of other programs, and eventually how to repair a bad resulting. The criteria are *attached* to operators (and goals).

There are common criteria attached to both complex and terminal operators:

- **Initialization criteria** to initialize values of input parameters.
- **Evaluation criteria** to assess the actual results of the selected operator after its execution. The operator results can often not be foreseen during planning but only determined after execution. Evaluation criteria detect and diagnose an impasse.
- **Repair criteria** let the expert express a strategy of repair and judgement propagation in a skeleton of operators. For example, the expert can express that the bad evaluation information should be transmitted to a sub-operator, or to its father operator, or to a particular previously executed operator.
- **Adjustment criteria** express a way to repair a problem (e.g. re-run an operator with modified parameter values) after a negative evaluation.

Example of adjustment rule (several examples of repair rules will be shown in section 4.2.1):

```
Adjustment criteria  
Rule name : r-adjust  
If  
    assess-operator? ambiguous  
Then  
    decrease parameter-1  
;
```

In the body of a complex operator an expert may define additional criteria:

- **Choice criteria** to select, among all the available [sub]operators, the operators which are the most pertinent, according to the data description, the context and the characteristics of the operators.
- **Optionality criteria** to decide if an optional operator must be applied depending on the dynamic state of the current data, domain objects, etc.

Example of choice rule:

Choice criteria

Rule name : r-choice

comment : "choice of operator constr-ch-with-filter if image is noisy"

Let ?c a Context

If

?c.noise == present,
size-filter > 0

Then

use-operator constr-ch-with-filter

;

3.3.3 Writing a Knowledge-base with YAKL

For a given set of programs, an expert describes the corresponding knowledge base in terms of *Operators*, *Goals*, etc., producing YAKL source. The YAKL source is then parsed, checked for consistency, and eventually translated into Lisp or C++ code. During the parsing of a knowledge base several syntactic and semantic verifications are done. For example, type checking in assignments, type compatibility between argument value type and default value or range, warning if parameters have no initialization means (default value or initialization rule), etc.

Depending on the engine, the same type of knowledge may not be used in the same way. For example preconditions of operators are simply tested before operator execution in HTN (Hierarchy Task Network) planners as OCAPI while they are used as clues for planning purposes in more dynamic planners as operator based planners like PHENIX.

Once the knowledge base has been written by an expert, the end-user can easily benefit of all the expertise about the utilization of a set of programs, through the resulting knowledge-based system. He/she only has to provide the knowledge-based system with information about the data instances of the current problem to solve, and a goal to achieve

(among those proposed by the system). The end-user can then concentrate on the application objective, without being disturbed by processing problems.

4 Dedicated Program Supervision Engines

In this section, we will describe three engines that we have developed for program supervision systems. Each engine is an improvement compared to the previous one. The first one (OCAPI), designed in 1990 [9, 11] is a HTN (Hierarchy Task Network) planner with a specialized execution module that allows some simple repair. The second engine (PÉGASE) is based on OCAPI, but we have extended its repair capabilities. The last one (PHÉNIX) is an attempt to mix the capabilities of a HTN and an operator-based planner. These engines use the same software development platform (see next section) and the same knowledge base language YAKL (see section 3.3). Because of this, we can use the same knowledge bases on these three engines without making significant changes¹.

4.1 OCAPI

OCAPI [9, 11] is the first program supervision engine that we have written. It is a HTN planner with a real execution module, with a well-defined evaluation functionality and a simple repair mechanism using only adjustment criteria. OCAPI was used for implementing the first version of the PROGAL knowledge base. *The originality of OCAPI is its ability to re-execute a plan in adjusting the parameters of the program depending of the evaluation results.* It works as a trial and error mechanism until the results are correct.

The evaluation in OCAPI operates automatically for quantitative evaluation and is performed manually (by the user) for qualitative evaluation. It also provides the possibility to use programs to evaluate other programs. OCAPI does not provide any help to construct these evaluation programs, but thanks to evaluation criteria a mechanism to use them when they exist. This evaluation mechanism is simple but works efficiently. The major problem lies in its planner, which does not permit replanning alternative plans when one fails. We have therefore added to OCAPI a better and more complex repair mechanism. This engine is called PÉGASE.

¹Generally, we need to add knowledge to fully use the engine capabilities. For example, to use a knowledge base developed for OCAPI with PÉGASE, we have to express repair criteria.

4.2 PÉGASE

The main advantage of a program supervision system as compared to a shell script is that it provides a repair module. If a failure or an error is detected, the program supervision system uses its repair knowledge to improve the results given by the plan. In order to express more complex repair strategies than in OCAPI we have developed PÉGASE which is a new engine based on the same program supervision model than in OCAPI but a richer repair mechanism. In PÉGASE, there are two main repair actions: re-execution action which takes the current plan and tries to find better values for tunable input parameters; and replanning action tries to find another plan. The repair method is given by the expert, when he/she models the program or the complex processing. A complete example of such mechanisms is given in [28]. The reader can also find interesting related work in [21]. PÉGASE is also a HTN planner[16] using the same structure and some mechanisms (evaluation and re-execution) used in OCAPI. Here is the algorithm used by PÉGASE:

- 0 **Ending** : if the resolution is finished.
- 1 **Planning** : if the current operator is a complex one:
 - **Initialization** of the operator using the initialization criteria (rule base).
 - **Decomposition** of the complex operator. If the link type is a:
 - **sequence** link, the sub-operators are scheduled to be processed.
 - **choice** link, call the choice criteria (rule base) to choose one sub-operator from the n possible sub-operators.
 - **optional** link, call the optional criteria (rule base) to decide if the sub-operator should be pushed to be processed.
 - **Execution of the effects** of the complex operator (rule base).
- 2 **Execution**: if the current operator is a terminal one.
 - **Initialization** of the operator using the initialization criteria (rule base).
 - **Preparation** of the operator (code generation to supervise the program)
 - **Test** if the pre conditions are true, otherwise **Error**.
 - **Execution** of the program using the correct calling syntax.
 - **Test** if the post conditions are true, otherwise **Error**.
 - **Execution of the effects** of the terminal operator (rule based).
- 3 **Evaluation** of the current operator (after its decomposition or after its execution)
Fire evaluation criteria (rule base) attached to the current operator. They mark the operator as correct or to be repaired.