Program Supervision : Yakl and Pegase+ Reference and User Manual

Sabine Moisan

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Program Supervision: YAKL *and* PEGASE+ *Reference and User Manual*

Sabine Moisan

N° 5066

December 2003

THÈME 3

ISSN 0249-6399 ISRN INRIA/RR--5066--FR+ENG

Program Supervision: YAKL and PEGASE+ Reference and User Manual

Sabine Moisan

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

Rapport de recherche n° 5066 — December 2003 — 187 pages

Abstract: This document describes concepts and tools that we have developed for the *program supervision* task. The first part is a reference manual, which introduces the concepts involved in this task as well as the proposed knowledge language and inference engine to achieve this task. The second part is a user's manual, which details both the YAKL description language and the PEGASE+ engine. It also provides examples and methodological recommendations to develop a program supervision system with YAKL and PEGASE+.

Key-words: program supervision, knowledge-based systems, knowledge description language, inference engine

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles, BP 93, 06902 Sophia Antipolis Cedex (France)
Téléphone : +33 4 92 38 77 77 — Télécopie : +33 4 92 38 77 65

Pilotage de programmes : manuel de référence et d'utilisation de YAKL et PEGASE+

Résumé : Ce document décrit les concepts et les outils que nous avons développés pour la tâche de *pilotage de programmes*. La première partie est un manuel de référence qui introduit les concepts de cette tâche, ainsi que le langage de description et le moteur d'inférence que nous proposons pour la réaliser. La seconde partie est un manuel d'utilisation qui détaille le langage YAKL et le moteur PEGASE+. Cette partie propose aussi des exemples et des recommendations méthodologiques pour développer un système de pilotage en utilisant YAKL et PEGASE+.

Mots-clés : pilotage de programmes, systèmes à base de connaissances, langage de description des connaissances, moteur d'inférence

Foreword

This document deals with concepts and tools in *program supervision.*

The *reference manual* introduces the concepts of the program supervision task as well as the proposed language and engine to achieve this task.

Program supervision is a task that is usually achieved by a human being, and which can be automated by software tools. This task implies to manipulate *concepts*, such as data, programs, sequences of programs, etc. The task is presented in chapter 1. We have gathered the useful concepts in an abstract general model which thus describes the "program supervision ontology". The model, detailed in chapter 2, also underlines the main reasoning phases that are applied while performing the task of program supervision. In order to describe and manipulate the abstract concepts, we need a tractable representation, *i.e.* some syntax. The syntax is a means to conveniently describe the concepts existing in the theoretical model. The YAKL language introduced in chapter 3, provides a syntax for the concepts in our model. It is easy to read for a human being and an automatic parser has been developed that translates it into computer data structures that can be managed by software tools. In our approach to program supervision the tools are knowledge-based systems *engines* that mimic the strategy of an expert in the use of the programs. An engine performs all the reasoning phases identified in the model. One particular engine, named PEGASE+, is detailed in chapter 4. Its algorithm is in charge of the reasoning and the controlling of the structures, according to the underlying model.

In the *user's manual* part, chapter 5 presents the full syntax of YAKL and the requirements for the parser/translator, and chapter 6 summarises methodological recommendations to develop a program supervision system with YAKL and PEGASE+. Chapter 7 illustrates on examples how the methodology can be applied and chapter 8 gives a simple complete example of a knowledge base. Chapter 9 gives an overview of the graphic user interface that is provided to the experts to browse and modify a knowledge base, as well as to trace an execution of a knowledge-based system. Finally, chapter 10 provides information about how to install and use all the tools. Two appendixes are added for more information about YAKL denotational semantics and about the list of possible error messages that may happen during parsing of YAKL source files.

Part I

Reference Manual

Chapter 1

The Program Supervision Problem

The use of existing libraries of programs has become a critical resource in many disciplines. Several 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 complex programs, written by specialists in one of these particular domains and often applied by non-specialists in these domains. To achieve a particular goal, several programs have to be organised in a plan and their execution has to be monitored in order to perform the best possible treatment. Using such libraries requires both extensive experience with the use of the programs and detailed knowledge of the overall task for which they are applied. Usually, given only the library of programs, a novice is unable to solve the complete processing task.

1.1 Motivations

Programs are often viewed as limited to source or object codes. But an effective long term management should also take into account attached knowledge as diverse as knowledge about the purpose, the scientific foundations, the intended applications, the conditions of applicability, the results of past tests, the know-how of everyday end users, etc. This versatile knowledge has been accumulated over the years (the life-time of the programs) and is scattered among different people, who are "sources" of knowledge about programs, either because they know the theory behind the code, because they have run the programs on numerous data, or because they have written or modified the code. Moreover, not only individual programs but also useful combinations of several programs to perform complex tasks are part of the knowledge. Even deep mining into the source code of programs does not provide enough information. Long term management and utilisation of such libraries requires different types of know-how ranging from an extensive experience with the day-to-

day use of the programs to a detailed knowledge of the physical process or mathematical structure they implement.

Indeed, the programs and their use belong to a company "patrimony" that should not be lost and that should be easy to re-use and to maintain. That is why companies need to keep track of all the necessary skills for the optimal use of programs, for both user assistance and knowledge management purposes. As an answer to this issue, we propose an approach based on:

- A conceptual model for experts, designers and users of programs allowing them to communicate about programs and their use with a unified terminology (based on generic concepts, such as data, programs, sequences of programs, data flow, etc.). Such concepts are recurrent and can be gathered in a *general ontology*;
- A descriptive *knowledge description language* to represent and manipulate abstract concepts. For this purpose, we have defined YAKL, an open language which provides experts with a user-friendly syntax and a well defined semantics for the concepts in our model;
- Computer *tools* to ensure the consistency of the expressed knowledge, to operationalise it into computer data structures and to produce effective systems to help run them (semi)automatically, provided that all the necessary knowledge is properly formalised

Techniques of program supervision have a twofold objective: both to favour the capitalisation of knowledge about the use of complex programs and to operationalise this utilisation for users not specialised in the domain.

The management of programs is generally performed by a person (termed the *expert* in the following) and it relies on a large amount of knowledge. Not only the code lines, but also the knowledge about how to run programs, how to evaluate their results, how to tune them, how to combine them for higher level computations, etc. is necessary. Thus, when experts -who have this know-how- are not available or when they retire or leave, it is necessary to keep this processing knowledge in an understandable and possibly operational form. Such knowledge is seldom made explicit in documentation and cannot be found in source codes[1].

1.2 Analysis of the Program Supervision Activity

When analysing the activity of using a number of complex programs for an applicative purpose, independently of the problem of the application (*i.e.* the goal of the user and the semantics of the data), it appears that a lot of problems come from the processing itself.

Given a set of data to process and a set of programs applicable on the data, the first point is to understand what each program does, *i.e.* build a model of the necessary concepts, such as programs, data, and so on. Afterwards, since a single program is not usually sufficient

[1] It should be noted that explicitly clarifying such knowledge also makes redesign and modification of codes easier.

to solve a complex processing request, the end-user must figure out which programs can be combined together and how. That means knowing how to choose which program should come first, then which ones may follow, and so on to eventually build "program combinations" that achieve an application goal.

Moreover, when multiple combinations are possible some can be preferred (for example, depending on the adequacy of program features with respect to the data at hand).

Then, to execute a chosen combination, the user has to actually run the programs, which implies knowing their precise calling syntax, together with their usual parameter values, the type of input they accept, and the type of output they produce (because the latter will become inputs for the following programs in a combination). Internal data-flow managing between programs may become very difficult to handle, e.g. if data are to be dispatched among different programs. Finally, if at any point of execution, the current results are not satisfactory, the user must infer which previously executed program is faulty, whether it can be re-run with new parameter values and how to compute the new values, or whether it must be replaced by another program.

Every end-user can not have such a deep understanding of the program semantics and syntax. One possible solution to this problem is to use a tool that transparently manages the processing complexity, in order to automate the easy reuse of the programs. Among different techniques for reuse, we propose program supervision techniques which aim at capturing the knowledge of program use in order to free the user from the processing details. The objective is to facilitate the automation of an existing processing activity, independently of any application. This means to automate the planning and the control of execution of programs (*e.g.*, existing in a library) to accomplish a processing objective, where each program computes one step of the processing. Using a program supervision system, a user's input request produces as output the executions of the appropriate programs with their resulting data.

1.2.1 Users of a Program Supervision System

A program supervision system typically addresses two kinds of users, experts and end-users. The expert knows the libraries' programs, so develops knowledge bases. For this purpose experts can use a problem solving environment containing the basic knowledge representation and reasoning structures. This manual is dedicated to experts building knowledge bases, using the problem-solving environment associated with PEGASE in LAMA.

An end-user can benefit of all the expertise about the utilisation of a set of programs, through the resulting knowledge-based system. He/she will actually use the system to solve a problem, without having any expertise about the algorithmic details of the programs. The only thing to do is to define the problem and the program supervision system solves it autonomously or advises what are possible solutions. The end-user can then concentrate on the application objective, without being distracted by processing problems. Note that the end-user is not necessarily a human-being but can also be a software module, as is the case with an autonomous system.

1.2.2 Our Approach to Program Supervision

Instead of developing each KBS from scratch, we design engines, independent of specific applications, but yet dedicated to the particular task of program supervision. A program supervision environment, associated with the engine, is used by a specialist of a library of programs to build a knowledge base. The result (*i.e.* the engine plus the knowledge base plus the library of programs) is a knowledge-based system for program supervision which can be utilised by an end-user to run an application. Program supervision is a very general problem, and program supervision techniques may be applied to any domain where complex processing is necessary and where each sub-processing corresponds to a suitable chain of several basic programs. To tackle this generality, we provide both knowledge models and software tools which are independent of any application and of any library of programs. We want them to be both general (*i.e.* independent of application domains and programs) and flexible, which means that the lack of certain type of knowledge has to be compensated by powerful control mechanisms, like sophisticated repair mechanisms. The goal of the model and of the proposed tools is to allows experts to "package" a raw code into an "informative module", which contains the knowledge on how to run it, how to evaluate its results, how to tune it, etc. It can be viewed as encapsulating programs, adding layers of different kind of knowledge to source codes: (*syntactic, strategic* and *semantic*), as shown in figure 1.1. Syntactic knowledge consists of calling syntax, order and type of input/output arguments, or even information such as operating system or memory required. Strategic knowledge corresponds to the way to assemble programs for complex tasks. Semantic knowledge is the specialist's know-how about the use of the programs and the decisions that should be made: e.g. what are the discriminant characteristics of a program, how to perform result evaluation or failure handling. Such packaging enhances the program with all the necessary knowledge to use and re-use it in different situations, to document it and to help maintain it. In this way, the specialist's reasoning knowledge may be (partially or fully) integrated. The result is understandable and reusable by other people in addition to the specialist who designed or implemented the code. Different categories of knowledge may be differentiated to perform program supervision: knowledge about the application domain, about the programs of a particular library as well as about the expertise domain (image processing for example) or about the problem solving strategy.

Such a module is understandable by other persons than the specialist who designed and implemented the code, and different modules can be connected in various ways for more complex tasks.

More formally, our model is based on a set of argument types and a set of operators. For a particular application, Ω denotes the set of available operators and Υ the set of their input and output argument types. Each operator $\omega \in \Omega$ is represented by: $(\mathcal{I}_\omega, \mathcal{O}_\omega, \mathcal{P}_\omega)$, where $\mathcal{I}_\omega$ (resp. $\mathcal{O}_\omega$) $\subset \Upsilon$ is the ordered set of types of ω input (resp. output) arguments and $\mathcal{P}_\omega$ the "protocol of use" related to ω (i.e. the semantic knowledge, in the form of a set of inference decisions to manipulate the operator). For the strategic knowledge, the model provides several composition operations: sequence, alternative, parallel, iteration, etc. to recursively organise operators into more abstract ones.

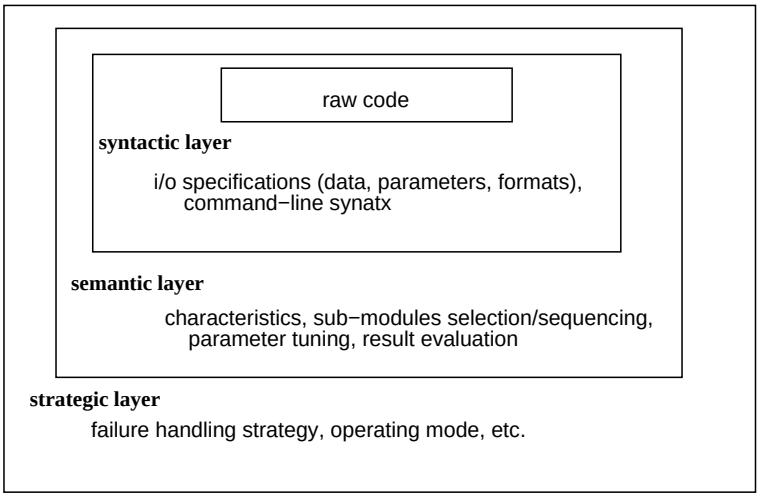


Figure 1.1: Program supervision provides a suitable framework for packaging raw code using different types of knowledge. Conceptually, codes are wrapped with layers of syntactic, semantic and strategic knowledge to form modules. This can be done in a recursive fashion for more complex tasks.

1.2.3 The LAMA Platform

To implement different program supervision systems, we have developed a framework, namely the LAMA environment [Moi98]. It is devoted both to knowledge base and inference engine design. It integrates ontological as well as problem-solving models. The task ontology corresponds to templates for knowledge base contents; it is implemented as a library of re-usable components (abstract classes) that can be derived when ontology extensions are needed. A knowledge base editor that supports YAKL is also provided by the platform. It is parametrised by the grammar of the language. An evolution of the syntax thus corresponds to a change in the grammar rules. Such an approach allows to reuse existing elements when possible, to extend them when necessary or to consistently add new ones without modifying the others. The library of re-usable components also provides instructions for writing the reasoning strategy for a program supervision engine as an algorithm tuned by a set of criteria. Additional tools are also provided in the environment, such as a knowledge verification toolkit adapted to the engine in use, a graphical interface both to visualise the contents of a knowledge base and to run the solving of a problem. Figure 1.2 shows the overall architecture of the LAMA environment.

The LAMA platform is generic and customisable. It allowed us to define three different program supervision engines, that propose variants of the general ontology and problem-solving mechanism and thus implied modifications of the ontology and of the syntax of the

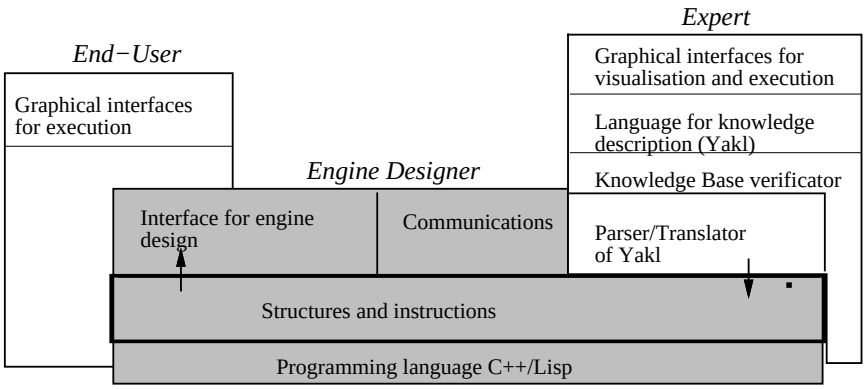


Figure 1.2: Architecture of LAMA and tools for engine designers, experts, and end-users

description language. These engines are briefly described hereafter and the PEGASE+ engine –which is the most used– is detailled in chapter 3.

The PEGASE+ engine is based on a hierarchical planning method, it was the first engine to introduce the concept of *optional* sub-operator in a sequential decomposition and the corresponding new kind of expert-defined criteria. Another important improvement concerns the failure handling mechanism, which introduces another kind of criteria (*Repair*).

The PULSAR [vdE96] engine combines hierarchical and dynamic operator-based planning methods. This second planning method matches the description of both the type and the contents of inputs and outputs with operator preconditions and effects. These concepts are thus better exploited than in PEGASE. In addition, PULSAR introduces *unordered decomposition*, a new type of composite operator decomposition and *weights* for attributes of argument types.

Finally, the MEDIA engine introduces additional concepts needed for its hybrid and perspective-based planning method, for example *perspectives* on data and *weak preconditions* on operators (preconditions that allow a better fit with data to be analysed and with objective, but can be relaxed when an optimal solution cannot be reached).

1.2.4 Formal Definition

More formally, we can define the program supervision process as follows:
Given as input:

- $\mathcal{P}$ ={ p_i/i ∈ 1..n } a set of programs p_i, (existing executable codes);
- {rp_i ∪ rc_j} a set of representations rp_i of the programs p_i and of their use, plus a (possibly empty) set of representations rc_j of known combinations c_j of the programs;
- {cr_k} a set of decision criteria;

- $\mathcal{I}$ a set of input data (real data, given by the end-user for a particular case);
- $\mathcal{EO}$ a set of expectations about output data (*e.g.,* their type and number);
- $\mathcal{C}(\mathcal{EO})$ a set of constraints on expected output data;

it produces as output:

- $\Pi = \{\{p_k \ / \ p_k \in \mathcal{P}$ and $\exists$ partial order on $p_k s \ \}\}$, a multiset, also named a *plan*, *i.e.* a combination of programs (with a correct data flow and where the same program may appear several times)
- $\mathcal{O}$ a set of actual output data such that:
  - $\mathcal{O} = \Pi(\mathcal{I})$,
  - $\mathcal{O}$ fulfils $\mathcal{EO}$ expectations,
  - $\mathcal{C}(\mathcal{O})$ holds.

1.3 Knowledge-Based Techniques for Program Supervision

Artificial intelligence techniques have been used to embody the expertise on the use of programs, in order to help a non-specialist user apply the programs in different working environments. Indeed, a knowledge-based system may manage the library use, freeing the user of doing so manually. This aid can range from advisory guide level up to fully automatic program monitoring systems.

Knowledge-based techniques that are used to build program supervision systems achieve both objectives of *operational problem-solving* and *knowledge capitalisation* about program use. Knowledge-based techniques allow the necessary expertise to be captured and stored for the support of a novice or an autonomous system performing program supervision. This know-how once integrated into a knowledge base makes this utilisation possible (even in robust automatic systems).

First, automatic and efficient operationalisation in the use of programs is necessary due to the large amount of existing programs that are run by end-users who belong to different domains. For example, image processing programs are more and more often used by physicians, biologists, astronomists, etc., or even in automatic processing chains. Program supervision systems offer non specialist users help concerning the choice, parametrising and sequencing of programs. They manage the selection of the best programs, their ordering, the data flows among programs, and they automatically detect problems and backtrack if necessary to perform repairs. They take into account the specifics of programs, the intermediate formatting of data, the complex calling syntaxes, etc. A program supervision system provides end-users with reconfiguration and adaptation capabilities, which are crucial

to the case of automatic processing chains. In some cases, a dialog may occur with the end-user -on some points relevant to his/her competence- in order to collect information to guide the reasoning process (typically to assess the quality of a result).

Second, program supervision techniques also allow experts to constitute a corporate memory about the use of programs that exist in their company. This fulfils a requirement for companies. Indeed, when experts -who have this know-how- are not available or when they retire or leave, the company wants to keep this processing knowledge in an understandable and possibly operational form. This knowledge is seldom explicited in documentation and cannot be found in source codes. Explicitly clarifying such a knowledge makes redesign and modification of codes easier.

1.4 Knowledge-Based Program Supervision System

A knowledge-based program supervision system emulates the strategy of an expert in the use of the programs. As any knowledge-based system it typically breaks up into a general inference engine, a knowledge base dedicated to a particular domain, and a fact base describing a specific problem in the domain (see figure 1.3). In the case of program supervision:

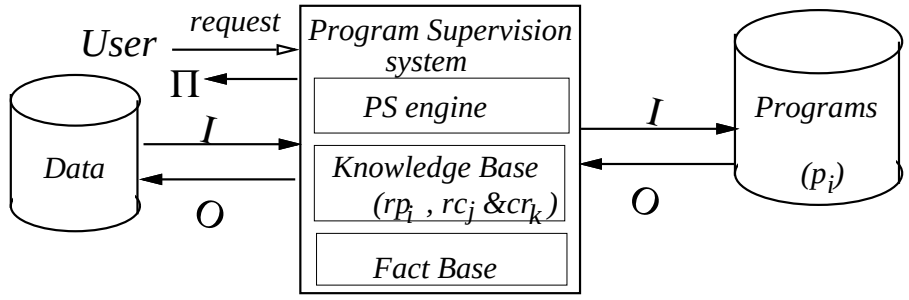


Figure 1.3: A knowledge-based program supervision system helps a user to use a set of programs for solving a request on input data $\mathcal{I}$ to obtain output data $\mathcal{O}$, as the results of the execution of a plan Π. It is composed of a program supervision engine and a knowledge base. The knowledge base contains the rp_i and rc_j representations of programs p_i and combinations of programs c_j, as well as the representations of various decision criteria cr_k.

- The program supervision (PS) engine is application-independent. It may use an auxiliary rule engine to process the different expert's criteria (expressed by rules in PEGASE) that are used during reasoning. The role of the engine is to use the knowledge stored in the knowledge base for effective planning, execution and control of execution of the programs in different working environments. It emulates the strategy of an expert in the use of programs. A supervision system is able to execute all the different treatment phases (more or less) automatically. Depending on the systems and on the application

domain the phases can be completely or only partly automated. To eventually obtain satisfactory outputs, the reasoning engine explores the different possibilities (different reasoning branches) and computes the best one, with respect to expert criteria, available in the knowledge base.

The usual engine cycle of reasoning phases to solve a user's problem is described in a general model of resolution in section 2.2).

- The knowledge base is written by an expert; it depends on the application domain and on the set of programs that is modelled. Achieving program supervision requires a clear description of the knowledge associated with programs and how they should be applied to solve a problem. The descriptions have to be as close as possible to the experts model, to facilitate knowledge acquisition. They should also be sufficient for the engine to select the programs, to initialise their parameters, to manage non trivial data-flow, and to combine the programs to produce a satisfactory plan depending on the input data, constraints, and request. The knowledge base encapsulates this expertise on programs and processing, *i.e.* knowledge about the correct use of a library of programs. This primarily includes descriptions of the programs and of their arguments, but also expertise on how to perform automatically different actions, such as initialisation of program parameters, assessment of program execution results, etc. Such automatic actions provide the final system with flexibility and robustness against changes of situations. A general model of knowledge for program supervision is presented in section 2.1. Note that even in a given domain there is no unique way of modeling the knowledge. See YAKL documentation for details on how to describe a knowledge base. It should be noted that during the reasoning process, the knowledge base is not modified.

- Once the knowledge-based system has been generated from the expert's knowledge base and the engine, the end-user only has to provide the system with information about the data instances of the current problem to solve and an abstract goal to achieve (among those accepted by the system). The fact base hence depends on the end-user's problem and contains the data instances describing this problem and all the necessary environmental information. During the reasoning of the knowledge-based system some data in the fact base may be modified or some other added (created as result of the execution of operators).

In addition to these usual components, a program supervision system includes the library of executable programs. Two kinds of interfaces are also added to this architecture: a (graphical) user-interface and a communication interface between the KBS and the library of programs.

Chapter 2

Program Supervision Model

We have identified the important concepts involved in a program supervision process. These concepts have been modelled from the point of view of software reuse as well as from the point of view of program supervision in data processing and artificial intelligence, in order to get the most widely usable representation. This chapter sketches the proposed model.

2.1 Proposed Knowledge Model in Program Supervision

The knowledge model defines the structure of program descriptions and what issues play a role in the composition of a solution using the programs. It is thus a guideline that enables to represent programs to be re-used and a guideline on how to re-use them. A description therefore should not only describe a program but also the information that is needed to apply it in different situations.

2.1.1 Model Ontology Summary

In this section we define the terms we will be using in this model. To achieve effective modelling of program supervision, we first define an ontology which contains general concepts, such as data or programs. This ontology provides experts with "patterns" (or reusable templates) that they will instantiate with respect to a domain[1], thereby obtaining a domain ontology. For example, if image processing is the domain, they may obtain an image-processing ontology containing the description of images and image processing programs. Refining a step further, they may even focus on a particular application, such as flaw detection; in this

[1] An application domain refers to the object (focus of cognition) of the programs, for instance mathematics or image processing are possible application domains.

case they obtain an application ontology, *e.g.*, containing the definition of artefact images and specific flaw detection programs.

We have identified the concepts that play a role in program use and modeled them in order to get the most widely usable representation, independently of any particular domain. As a result we propose guidelines that enable the representation of programs and issues that play a role in the composition of a solution using the programs. It is thus also a guide on how to (re-)use them. Since the model presented here is intended to be independent of any implementation, we have chosen general terms to name the common concepts involved in program supervision. The terminology we have chosen is the result of an analysis [vdEvHT95, CMM98] of many existing systems related to program management that we have either developed or closely studied. Even if each system has its own vocabulary, some terms (like "operator") are widely used. The next sections define the concepts of the proposed general ontology and their concrete representation in YAKL. The most important concepts are the *operators*, with their *arguments* and attached *criteria*.

- *Operators* (see section 2.1.2) perform actions and manipulate data.
  - Primitive operators correspond to programs;
  - Composite operator correspond to known combinations of operators that solve abstract processing step;
- *Arguments* (see section 2.1.3) are attributes of supervision operators.
- Attached to operators, various *expert criteria* (see section 2.1.4) are used to describe decisions during problem solving.
- *Data and domain objects* (see section 2.1.5) contain all necessary information on the problem of the end-user.
- Abstract *functionality* expresses an objective to achieve (see section 2.1.6).
- A *request* (see section 2.1.6) express a user's query, *i.e.* a functionality to achieve and the data of the particular case to work on.

All these concepts are interrelated (see figure 2.4) and will be used by the problem solving mechanism (see section 2.2) of a system, to produce as result a *plan* (see section 1.2.4 and figure 1.3), consisting of a partially ordered list of (primitive) operators to be executed.

This approach provides experts with guidance for knowledge representation. The ontology helps making explicit the role of knowledge elements in program supervision (such as parameters) and allows them to identify missing or irrelevant knowledge (for instance lack of argument setting).

The following sections detail the concepts in the ontology.

2.1.2 Supervision Operators

Supervision operators are used to define elements which perform actions and manipulate data. They represent either concrete programs (primitive operators) or abstract processing (composite operators). Both have input and output arguments (data or parameters). Both kinds of operators also encapsulate various criteria (which may be represented by rule bases) in order to manage their input parameter values (initialisation criteria), to assess the degree of quality of their results (evaluation criteria on output data), or to react in case of bad results (repair criteria). Several operators (of both types) may have to be applied to achieve one single user's abstract processing.

Their common representation includes (most items are optional):

- An optional reference to an abstract functionality –or processing objective– through the name of a `Functionality` object previously defined, *i.e.* information on "what is the operator for?" (*e.g.* "factor_estimation", or "segmentation" may be defined as functionalities in image processing). A connection with a functionality is only necessary for those operators that will be able to answer a user's request.
- Optional characteristics: a symbol list describing non functional characteristics of an operator, known by the expert (*e.g.* "slow, resource_consuming"),
- Information on arguments, including their names, types, ranges or means to compute their value (for input and output arguments, *i.e.* "on what does it acts?");
- Pre and post conditions are tests which have to be checked before and after the execution of an operator. Preconditions apply on input data and state whether the operator is applicable. Postconditions apply on output data and state what should hold after the application of the operator. They may be used as clues during dynamic planning; conditions are described by expressions referring to slot values of data arguments (or domain objects);
- Expected effects describe what the operator achieves and what its side effects are on the output, after execution, independently of which data the operator is applied on.
- Various criteria to specify the reasoning which is made on operators, such as initialisation and adjustment of arguments or evaluation of the performance of the operator (correctness of the results).

Supervision operators representing concrete programs are referred to as *primitive operators*. In addition to this common information, primitive operator descriptions contains all the information needed for the effective execution of the program (including its calling syntax). The execution of a primitive operator corresponds to the execution of its associated program, provided that its execution conditions are true. An abstract view of two primitive operators is shown in figure 2.1.

Composite operators are the representations of higher level operations. They don't have attached operational actions but they break down into more and more concrete (composite

Primitives rp_i		
Functionality	image thresholding	alignment of genomic sequences
Input		
Data	image1 to threshold	. genomic sequence . genomic bank
Parameters	threshold	. percentage of identification . coeff. of matching
Output	image2 thresholded	resulting file
Preconditions	noise image1 =Gaussian	
Call	cd *image1*.path; muls -s *threshold image2*	blast *bank sequence* *percentage...*

Figure 2.1: Abstract view of primitive operators in two domains: image processing and genomic analysis. A primitive operator may achieve a functionality (*e.g.*, thresholding). it has input and output arguments, and preconditions, in order to execute (*e.g.*, on input data format). Its calling syntax will be instantiated at execution time with the actual values of arguments.

or primitive) sub-operators. They therefore correspond to decompositions that are usually predefined by the expert in the knowledge base. The usual types of decompositions are specialisation (or alternative), sequence, parallel, and iteration. In all cases the sub-operators in a decomposition may in turn be either primitives or composite ones. Since several operators can concretely realise one abstract functionality the specialisation decomposition type provides a way of grouping operators into semantic groups corresponding to the common functionality they achieve. This is a natural way of expression for many experts because it allows levels of abstraction above the level of specific operators. The existence of these alternatives leads to a richer and more flexible knowledge base, with a wider range of applicability. In a sequential decomposition some sub-operators may be optional. These decompositions -at different levels of abstraction- must end with primitive operators. A composite operator is therefore refined into its sub-components. In addition to the common information, the way to refine a composite operator is expressed by:

- Control information about the type of decomposition into sub-operators,
- References to the sub-operators (e.g., by their names)
- Data flow information between a father operator and its sons and data flow between sons in a sequential decomposition.
- Additional criteria (for choices, optional applications of sub-operators, and repair strategy)

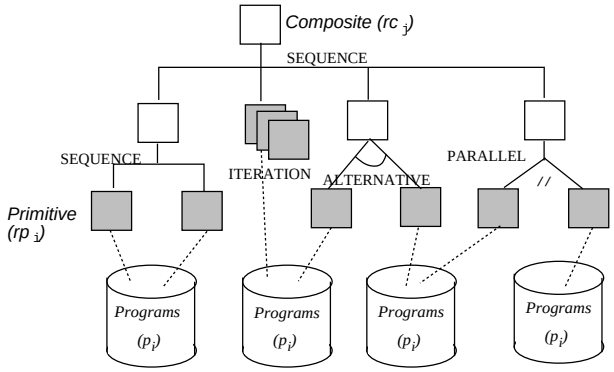


Figure 2.2: A composite operator and its sub-operators. Composite operators are represented by white squares, while primitive ones are represented by grey squares ones. In this example there is a sequence at first level, composed of another sequence, an iteration, an alternative, and a parrallel. Primitive operators encapsulate programs, which may belong to different libraries.

2.1.3 Arguments

Arguments are associated with operators (primitive or composite) and with functionalities. They play an important role because many decisions (e.g., the selection of a program) are based on the information that arguments provide. This is particularly true if processing is data-driven, as in image processing. An operator with an attached functionality must have at least the same number of arguments, with the same type (and for the moment also the same names) as its functionality, but it may have additional ones (more parameters, for example).

We differentiate two categories of arguments, data and parameter arguments. Data arguments have fixed values which are set for input data (*e.g.*, an input raw image), or computed for output data (*e.g.* an output segmented image). Parameter arguments are tuneable, *i.e.* their values can be set by means of initialisation criteria or modified by means of repair criteria. Parameters are always input arguments. The initial input data are provided to the system by the user in an initial *request*.

Output data arguments are computed during the reasoning process All arguments (but preferably output data) can be "assessed" by means of evaluation criteria, and these judgements drive the process of result evaluation and parameter adjustment.

2.1.4 Criteria

Finally, different types of criteria can be attached to operators. There are common criteria attached to both composite and primitive operators (initialisation, evaluation, adjustment and repair). There are also additional criteria defined for a composite operator (choice, op-

tionality). The criteria are used to decide how to choose among alternatives, how to initialise input arguments, how to evaluate results (output data), how to adjust the processing with the determination of new input values for programs or the selection of other programs, and how to repair in case of bad results. Criteria provide a program supervision system with flexible reasoning facilities.

Common criteria

These are criteria common to composite and primitive operators. For each operator an expert may define four kinds of criteria.

- Initialisation criteria contain information on how to initialise values of input arguments, before executing the current operator.
- Evaluation/assessment criteria state the information on how to assess the quality of the selected operator's actual results after its execution. The operator results could not be foreseen during planning but only determined after execution. Evaluation criteria allow to detect and diagnose a problem[2].
- Adjustment criteria express a way to locally repair a problem by re-running an operator with modified parameter values, after a negative evaluation. Parameter adaptation can be performed by any method provided by the expert (see page 46 for details in PEGASE+).
- Several trials and errors are often necessary to obtain correct final results. A failure handling mechanism (or repair mechanism) is necessary to fix the possible problems. It is performed either locally inside an operator (by adjustment criteria in PEGASE) or non-locally by message transmission to another operator in the plan. In PEGASE+ repair criteria, along with adjustment criteria, are assumed to describe a complete strategy of repair and of diagnosis propagation in a hierarchy of operators after a negative evaluation (see page 45 for details in PEGASE+).

Criteria for composite operators

For a composite operator an expert may define other specific criteria:

- Choice criteria are attached to composite operators with a specialisation decomposition type. They select the operator(s) which is (are) the most pertinent among all the available sub-operators, according to the data descriptions and the characteristics of the operators. This kind of criteria is used for planning purposes.
- Optionality criteria are attached to composite operators with a sequential decomposition type with some optional sub-operator(s). They decide if an optional sub-operator has to be applied depending on the dynamic state of the current data.

[2] In PEGASE the diagnosis is expressed as a symbolic judgement, usually on a parameter value.

Figure 2.3 shows a high-level view of the main types of criteria, with their typical conditions and actions. Several rules of each type constitute a rule base that contains the expert's know-how on particular reasoning decisions. These are only examples and many other types of conditions and actions are possible for each kind of rule (see YAKL reference manual chapter for details).

Typical rules for criteria	
Choice	Initialisation
If Object attribute *a* has value *v* Then Use program *p*	If Object attribute *a* has value *v* Then Set parameter *p* to value *v1*
Assessment	Repair
If Result *r* has property *p* Then Declare problem *pb* for *o2*	If Operator *o1* has problem *pb* Then Transmit problem *pb* to operator *o2*

Figure 2.3: Abstract view of the most usual criteria, shown in the form of inference rules (with a pseudo natural language syntax). Several rules of each type constitute a rule base that represents the experts' criteria.

2.1.5 Data and Domain Objects

Data and domain objects correspond to the description of data used by the programs and of some application domain concepts which may influence the program supervision process.

Data and domain objects are stored in a base of facts (fact base) and they contain all necessary information regarding the user's program supervision problem. The data description in the knowledge base plays an important role in program supervision because many decisions are based on the information that data provide. This is particularly true if processing is data-driven as in image processing, for example.

Concerning domain object, the purpose is not to provide a complete domain description of objects involved in the application, but only of those which could influence the supervision process. The domain objects are highly dependent on the application domain.

2.1.6 Functionalities and Requests

A functionality allows to group together all the operators achieving the same abstract function (*i.e.* "thresholding" in image processing). Its description only mention compulsory inputs and outputs to perform the function. Requests are queries for an abstract functionality on particular data, under particular constraints. They are the means for the end-user of

a knowledge-based system to describe the initial problem. The aim of a program supervision knowledge-based system is to respond to requests coming either from human end-users or from another software system.

2.1.7 Interrelations of Concepts

The ontology not only defines the concepts but also their relationships. Figure 2.4 summarises the main concepts of the model that have been presented in the previous subsections and shows their relationships. It is an abstract, simplified view in the form of an UML (Unified Modelling Language) class diagram.

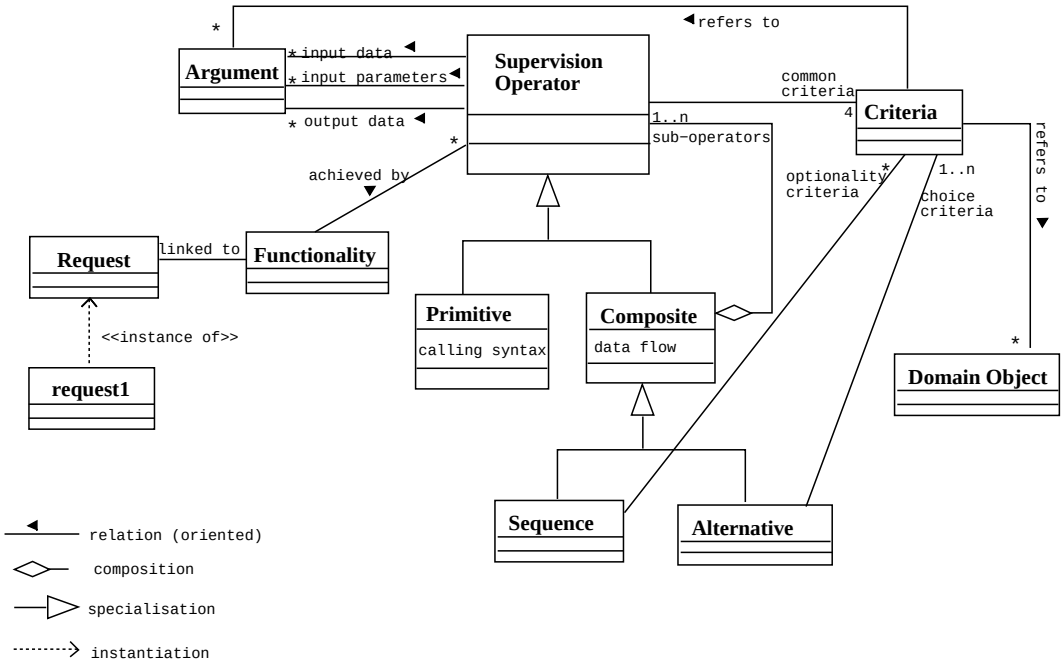


Figure 2.4: Relations between concepts in the ontology.

Most of the relations are "one to many", *e.g.*, one supervision operator is connected to many input data (in UML notation, * denotes 0 or more) and to 0 up to 4 common criteria (as presented in section 2.1.4): since no criteria of any kind is compulsory, an operator may have no criteria at all. In the same way, a composite operator is connected by a composition relation (denoted by a diamond) to several (at least one) sub-operators. Not all the possible types of composite operators with respect to the type of their decomposition are represented in this simplified view (only sequence and alternative types are because they are the most commonly used). A primitive operator is a leaf of the hierarchy.A request and a functionality

are connected together. A functionality is itself connected to all operators that can achieve it, because a functionality corresponds to an abstract processing objective and may therefore be achieved by several operators. Each operator has in/out arguments and attached criteria (which may refer to the operator arguments and/or to significant domain objects).

2.2 Model of Problem-Solving Mechanism

All introduced concepts are managed by a problem-solving mechanism, which is also dedicated to the program supervision task. In parallel with the specification of the general ontology described above, we have developed a general problem-solving mechanism (or method) for program supervision. This mechanism is implemented in a program supervision engine. The role of the engine is to exploit knowledge on programs in order to produce a plan of programs that solves the user's problem (as shown in figure 1.3). It mimics the strategy of an expert in the use of programs. The reasoning engine explores the different possibilities and computes the best one, with respect to available concepts descriptions.

First, the solving of a program supervision problem starts with a user's request, which states the functionality, the data on which this is to be achieved, and the context (some domain object values) in which the problem is being solved. Then the problem-solving mechanism of a program supervision system's engine is launched. This mechanism emulates the strategy of an expert in the running of the programs. Different techniques may be adopted for that purpose, most of them are based on artificial intelligence planning techniques. It is the case in PEGASE+.

A general model of this mechanism can roughly be decomposed into several phases, as shown in figure 2.5:

1. Preliminary problem *identification* in term of a functionality to achieve. This phase analyses the user's request in order to translate it into a program supervision concept.

2. *Construction* proposal (selection and rank-ordering of programs, based on composite operators, arguments, pre/postconditions and effects). This phase selects and assembles programs, in order to build a (partial) executable series of programs (or even to generate a code in some systems). It decides on the order of program executions to solve the user's problem on particular data.

3. Effective *execution* of codes (based on primitive operator descriptions). This phase runs (directly or via a communication protocol) the programs that have been ordered by the previous phase after having filled current data and adequate parameter values. It produces results.

4. *Evaluation* of result quality (by evaluation criteria). This phase which detects potential problems concerning the returned results (depending on the systems a problem may be an execution error, a lack of information, an incomplete data, or a poor quality for a result). It may be performed either in interaction with the end-user or it can

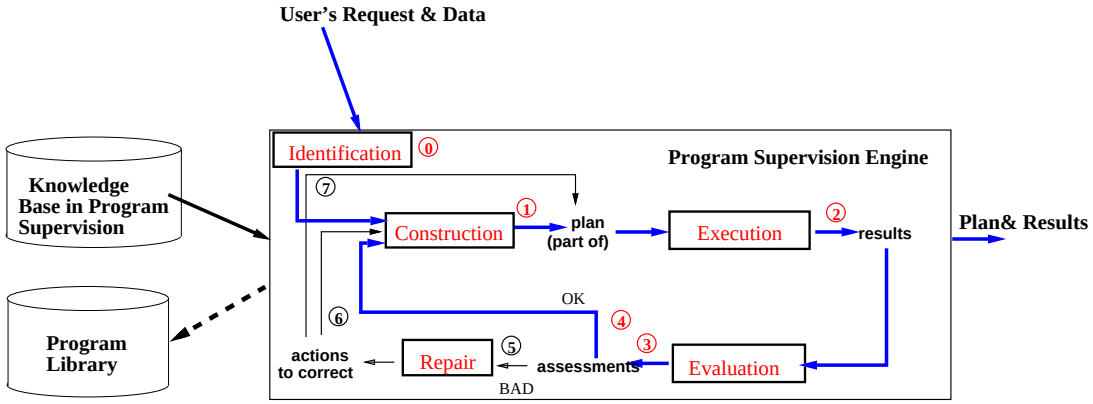


Figure 2.5: Main phases of program supervision: *identification* transforms the user's request into a PS objective, then *construction* selects and rank-orders the processing programs, leading to a (partial) plan (1), *execution* calls the actual programs and produces results (2) that are passed to *evaluation*, which returns assessments (3). If the assessments are correct (4), the planning process can go on. If failures have been detected (5), *repair* decides which correcting actions are appropriate (e.g. replanning (6) or re-execution (7)) after tuning some parameters. Blue bold arrows show the main recursive loop. Plain black arrows show the repair loops.

sometimes be fully automated thanks to measurement computation, expert-defined criteria, methods to compare results with reference cases, etc, which are part of the knowledge base. This phase produces assessments on results. If the results are good enough, the process can go on.

5. If the assessment on results is negative, it is considered as a failure and a *repair* phase decides which corrective actions are appropriate to undergo with respect to the current proposal and the current problem (using repair and adjustment criteria). This may lead to either reconsidering the arrangement of programs or changing some parameter values. This phase may be automatic or interactive. Note that the failure detection and repair mechanisms stay at the level of program supervision expertise. That is to say that a failure is detected only when assessing the *results* of a program execution and repaired only if the expert has been able to provide knowledge on how to repair it. If the failure is caused by the program itself (*e.g.*, an incorrect algorithm) or by hardware problems, program supervision will of course be of no help to repair it.

Each phase relies on the semantics of the knowledge it uses. For example, in phase 3, a prerequisite to executing a program is to initialise the values of its parameters: it is the role of the initialisation criteria in our model. We have defined a denotational semantics for all concepts in the general ontology (see annex A.1).

The process stops when either a proposal has been accepted as a solution by the user or by the system (via an expert criteria) or when the system cannot make any other proposal.

According to the type of reasoning process to carry out, there may exist *variants* of the phases of this general model, that can also be more or less interleaved. For instance, the construction phase may be based on a planning Hierarchical Task Network (HTN) or on operator-based planning techniques. Some systems offer a pure construction phase (often based on planning techniques) and a postponed execution. When an evaluation/repair mechanism exists all the phases are often interleaved because the effects of evaluation and repair must be taken into account during the construction and each construction step may depend on information that is only available after the execution of previous programs in the plan. At a lower level, some basic steps can be performed in a more or less complex way.

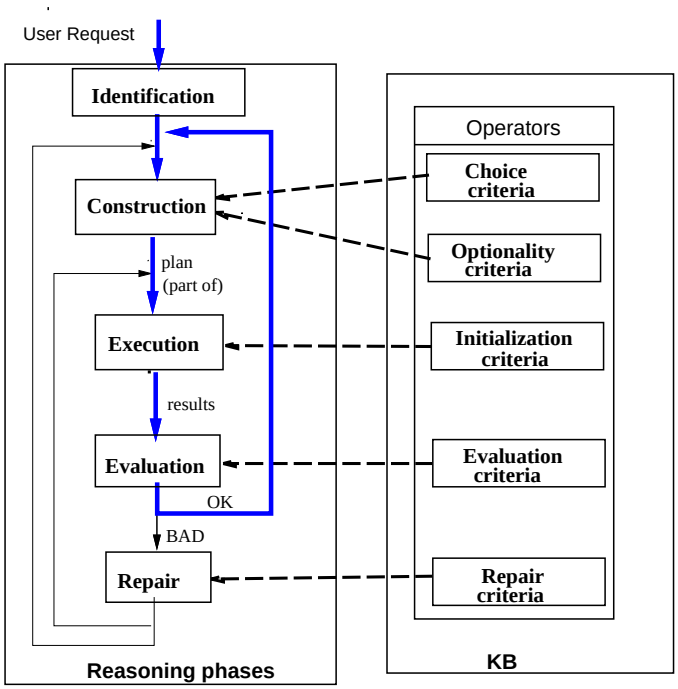


Figure 2.6: Relationships between decision criteria and reasoning phases. Dotted arrows show which type of knowledge base (KB) criteria is used for which reasoning phase.

During the reasoning phases, the problem-solving mechanism exploits the concepts descriptions. All the concepts and especially the decision criteria play different roles and they are involved in different phases of the problem-solving mechanism in the program supervision engine. Depending on the engine, the same type of knowledge may not be used the same

way. For example, preconditions of operators are simply tested in PEGASE before operator execution while they are used as clues for planning purposes in other planning strategies.

Concerning decision criteria, figure 2.6 summaries their relationships with the different phases of the reasoning. For example, the execution phase uses the knowledge about the concrete syntax of the programs and the initialisation criteria, while choice criteria between supervision operators are related to the construction phase, and the repair phase uses experts' repair criteria in cooperation with the engine mechanisms. The richer the knowledge base is in terms of criteria, the more flexible the related reasoning phase will be.

2.2.1 Formalisation

More formally, we can define the program supervision process as follows:
Given as input:

- $\mathcal{P}$ ={ p_i/i ∈ 1..n } a set of programs p_i, (existing executable codes);
- {op_i ∪ oc_j} a set of so-called operators representing the programs p_i and their use (op_i), plus a (possibly empty) set of operators oc_j representing known combinations of the programs;
- $\mathcal{I}$ a set of input data (real data, given by the end-user for a particula r case);
- $\mathcal{EO}$ a set of expected output data (only their type and number are known);
- $\mathcal{C}(\mathcal{EO})$ a set of constraints on expected output data;

it produces as output:

- Π ={p_k /k ∈ 1..m, m <= n, p_k ∈ $\mathcal{P}$ and ∃ partial order on p_ks }, a plan i.e. a combination of programs
- $\mathcal{O}$ a set of actual output data such that:
  - $\mathcal{O}$= $\Pi(\mathcal{I})$ and
  - $\mathcal{C}(\mathcal{O})$ holds.

Chapter 3

Introduction to the YAKL Language

The objective of YAKL is to provide a concrete means to capitalise in a both formal and readable form the necessary skills for the optimal use of programs, for user assistance, documentation, and knowledge management in a company. First, a readable syntax facilitates communication among people (*e.g.*, for documenting programs) and, second, a formal language facilitates the translation of abstract concepts into computer structures that can be managed by software tools.

YAKL models only what is relevant to communicate about programs and to manipulate (and run) them, without exposing their code. For this purpose, we define *representations* of programs and we provide *composition operations*, such as sequence or alternative, to produce higher-level combinations for complex tasks. In the following, we use the general term of *operator* for the representations of both real programs and combinations.

3.1 Elements of the Language

Based on the general ontology, we design the YAKL language that offers a concrete syntax to describe the concepts of the abstract model, at the right level for each development role. It provides programmers with a way to document their programs, technicians with a way to note guidelines to appropriately use programs, scientists with a way to annotate programs and to link them with formulae or theoretical papers (and vice versa i.e. to find programs connected with the same theory). YAKL is a means to describe the knowledge about a set of programs, independently of any implementation language, any domain, or any application. It is used both as a common storage format for knowledge and as a human readable format

for writing, exchanging, and consulting knowledge. We have in parallel defined a formal semantics for the language (see appendix A.1).

From an operational point of view, the language can also be translated into computer structures to produce an executable knowledge-based system. YAKL already captures most knowledge about program use, even elements which are seldom explicited in other approaches (e.g., repair strategy). Furthermore it is an *open* language that can be extended or adapted to suit different needs.

It should be noted that the language provides a syntax uniquely for the general ontology (referring to common concepts, such as "program", "argument", etc.). Based on it, experts can build knowledge bases to define and use other kinds of ontologies (domain and application ontologies mentioned before), which are out of the scope of YAKL (e.g., an image processing ontology, referring to image processing concepts).

YAKL uses both frame-based and rule-oriented descriptions. Frames are used for operators or arguments, whereas inference rules are used for criteria.

3.1.1 Operators and Arguments

Operators represent either concrete programs or abstract processing. Both have the same common information about their input and output arguments and both encapsulate various criteria in order to manage their input parameter values (initialisation criteria), to assess the degree of quality of their results (evaluation criteria on output data), or to react in case of poor results (repair criteria). Several operators (of both types) may have to be applied to achieve a single user's abstract processing.

Arguments are represented as operator attributes. YAKL provides them with a frame-based representation and a hierarchical organisation for argument types.

For instance the YAKL source to define a new argument type (`Polynom`) for mathematical processing is defined below, simply as an extension of a file (containing the text of a polynomial system, plus slots containing information about numbers of variables and of equations). This type will be added to Υ and it can be used latter to type operator arguments. YAKL syntax is close to natural expression, but more structured (keywords are indicated in bold face). In particular, the frame slots have several optional predefined facets (e.g. *range* or *default*).

Argument Type { name Polynom **Subtype Of** File
  Attributes
  Integer **name** nb_variables
    default 1
  Integer **name** nb_equations
    default 2 **}**

Operators representing concrete programs are referred to as *primitive operators*. They describe the programs as "black boxes" known only by information on how they can be used in different situations and by their input and output arguments. In addition to the common information, their descriptions contain the information needed for effective execution of programs (including calling

syntax). The execution of a primitive operator corresponds to the execution of its associated program.

The structural part of YAKL code to describe an image thresholding operator is given below (we suppose that a type `Image` has been previously defined, with a `noise` attribute). It details the achieved functionality (thresholding), input and output arguments, a precondition on image noise, and the calling syntax, which has to be instantiated at execution time with the actual values of arguments.

Primitive { name thresh
Functionality thresholding
Input Data
Image **name** image1 **comment** "original image"
Input Parameters
Float **name** threshold
default 1
Output Data
Image **name** image2 **comment** "thresholded image"
....
Preconditions image1.noise == gaussian
(Criteria omitted ... see 3.1.2)
Call
language shell
syntax cd image1.path ";" thresh -s threshold image2 **}**

Composite operators represent higher level operations. They break down into more and more concrete (composite or primitive) sub-operators. They therefore correspond to useful decompositions that are predefined by the expert. These decompositions at different levels of abstraction must end with primitive operators. Currently, YAKL offers alternative (|), sequence (-), parallel (||), and iterative (*) as types of decomposition. In a sequential decomposition some sub-operators may be optional. Alternative decompositions provide a way of grouping operators into semantic groups corresponding to the common functionality they achieve. This is a natural way of expression for many experts because it allows levels of abstraction above the level of programs. In addition to the common information, the way to refine a composite operator is expressed by:

- Control information about the type of decomposition into sub-operators,
- References to sub-operators,
- Data flow information between a parent operator and its children (and between children operators in a sequential decomposition),
- Additional criteria (for choices, optional applications of sub-operators in a sequence, and repair strategy).

The YAKL frame corresponding to the composite operator `multivar` of figure 3.1 is presented below. It uses the previously defined argument type `Polynom`.

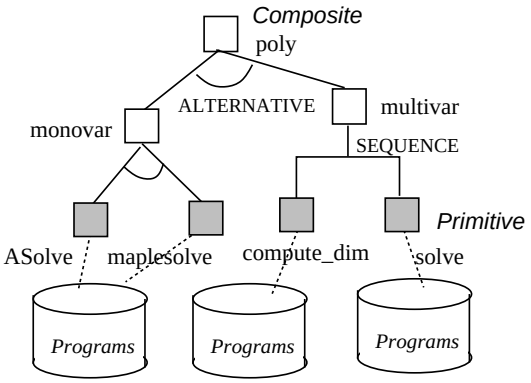


Figure 3.1: Example of mathematical operators. Composite ones are represented by white squares, primitive ones by grey squares. In this example, there is an alternative at first level, composed of another alternative and a sequence.

Composite Operator { name multivar
comment "solve polynomial systems with nb. variables >1"
Input Data Polynom **name** Sy
Output Data Polynom **name** sol
Preconditions Sy.nb_variables > 1
Body compute_dim **-** solver *(- stands for a sequence)*
Distribution *(data flow parent-children)*
multivar.Sy / compute_dim.PSin
multivar.sol / solve.sol
Flow compute_dim.PSout / solve.Sy *(data flow among children)* **}**

3.1.2 Criteria

Different types of criteria can be attached to operators: common criteria attached to both composite and primitive ones and additional criteria for composite operators only. Criteria represent the dynamic knowledge about *decisions* (e.g., how to choose among alternatives or how to adjust the processing with the determination of new input values for programs or the selection of other programs). Criteria provide a system with flexible reasoning facilities. For the time being, the criteria are represented in YAKL by specialised rule bases (groups of rules) which are attached to operators. Initialisation of parameters, result evaluation, repair and adjustment rule bases can be attached to all operators, while choice and optional criteria are specific to composite ones. The locality of the criteria allows each piece of knowledge to carry its own decision knowledge with respect to its role in the processing and the kind of information it has access to.

The external form of a all kinds of rules is:

Let *declarations*
If *premise*
Then *conclusion*

which means: "if the situation described by the premise, after correct instanciations of declarations, is recognised, the rule engine will launch the corresponding conclusion".

The *declarations* part declares typed free variables that could be used in premise or conclusion. Free variables refer to object in the fact base and their types are types of the fact base (domain object usually). Regular local variables can also be declared.

The *premise* part checks some properties of global variables or of the free variables in declarations that correspond to a typical situation with respect to the expert knowledge. Several combinations of objects may verify the premise conditions (these combinations are called instantiations of the rule).

The *conclusion* part is composed of actions that have to be performed if the situation has been recognized by the premise. They consist of object slot value modifications of objects in the fact base, creation or destruction of objects, or, in program supervision, argument modifications of supervision operators. Actions can also modify the control of reasoning (*e.g.*, decide to stop executing an operator) (see chapters on YAKL for syntax details).

As an example, below is the YAKL form of the choice criteria of operator `poly`, that decides whether to choose `multivar` or its alternative `monovar`:

Choice criteria
Rule { name choice_mono
If PS.nb_variables == 1
Then use_operator monovar **}**
Rule { name choice_multi
If PS.nb_variables > 1
Then use_operator multivar**}**

3.2 Use of the Language

All concepts of our ontology are to be managed by inference mechanisms relying on knowledge description. YAKL provides experts with a framework to store their knowledge about programs in a knowledge base. Such a knowledge base contains only information that plays a role in program management, i.e. descriptions of operators, their arguments, their competence and applicability conditions of operators, their relations, etc.

The YAKL source of a knowledge base can be parsed and eventually translated into various formats to be processed by computer tools: an inference engine for execution, a graphical tool for visualisation, a simulator, etc. During the parsing, syntactic and semantic verifications are performed on the knowledge description: e.g., type checking in assignments, type compatibility between argument value type and default value or range, warning if parameters have no initialisation method (default value or initialisation criteria), etc.

Note that a knowledge base designed with YAKL can potentially be run by several inference mechanisms, provided that it contains the required information for the inferences. Different inference mechanisms may not use the same knowledge parts or not in the same way.

The YAKL language has been designed to offer a model-based view of the use of programs which is easy to comprehend because it conceals implementation or domain-dependent details. Using this language, experts can express their knowledge at the expertise level, guided by dedicated representation patterns provided by the underlying ontology. YAKL offers a high descriptive power convenient for most applications (except for distributed features) and can be *adapted* to various needs. Its human-readable form is easily adopted by non computer scientists, yet it can also be translated into various formats (e.g. XML) in order to facilitate its interoperability with existing tools (e.g. on the Web). It helps *formalise* the description by providing a common language to experts, which is understandable across domains, with a formal semantics. Thus it enables *sharing* of knowledge between experts. Moreover, it can be used in an incremental manner: from simple code documentation to a real knowledge base for a program supervision system.

Chapter 4

Inside PEGASE+

PEGASE+ is an engine dedicated to program supervision knowledge-based systems, which can automate the choice and execution of programs from a library to accomplish a processing objective. The PEGASE+ engine provides a HTN (hierarchical-task network) planner, an execution module, some evaluation facilities and a repair mechanism using repair and adjustment criteria. It works as a trial and error mechanism until the results are correct.

PEGASE+ is currently the most developed and tested engine that we have. Since 1994, several knowledge bases have been developed with PEGASE+ in different application domains [TMC99]. For example, in image processing for automatic target recognition [SMV+99], in medical imaging [CAM+97], or for satellites images [MMMV96].

4.1 Components of a PEGASE+ KBS

4.1.1 Engine

Following the general model presented in the first chapter, the PEGASE+ engine is in charge of automatic planning, execution and control of the programs, based on the knowledge base contents. The algorithm of PEGASE+ is based on hierarchical planning (HTN). Given as input a user's request on particular data, PEGASE+ selects the most appropriate path (depending on the context of application) in its hierarchy of operators (representing possible plans). The final plan that produces satisfactory outputs is usually not straightforward, it often results from several trials and errors, performed automatically by the engine. For this purpose the engine builds a state tree, to be used to backtrack on bad decisions.

A general view of the algorithm of the PEGASE+ engine is shown in figure 4.1, and is detailed in section 4.2. An ancilliary rule engine in forward chaining is called each time a

criterion is triggered. This engine is the same for all types of rule and all the rules have the same general syntax.

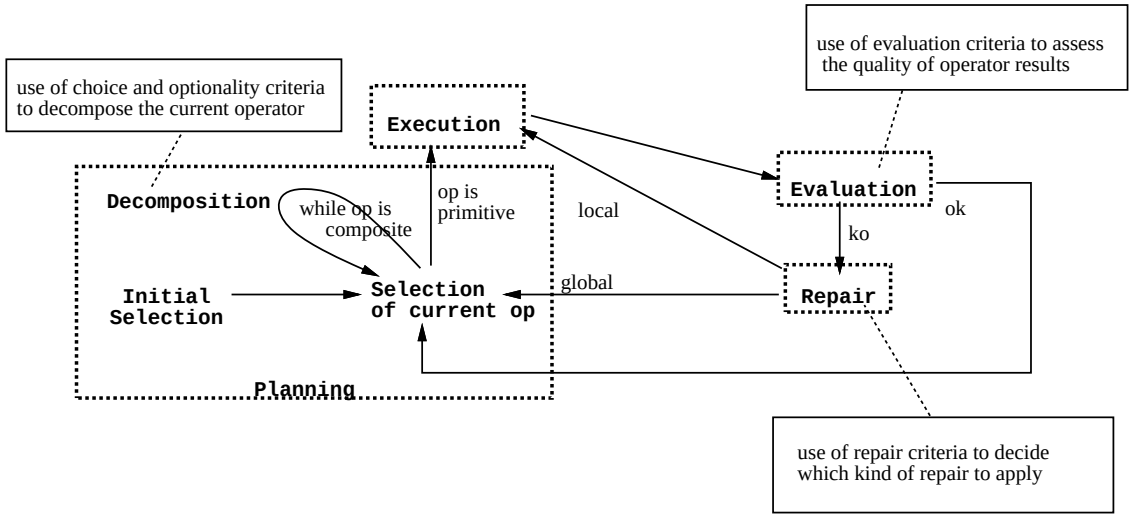


Figure 4.1: High-level view of PEGASE+ algorithm

4.1.2 Knowledge Base

A *knowledge base* for PEGASE+ mainly contains *operators* which are representations of programs (with descriptions of their data and parameters) and of typical combinations of programs, as well as criteria to guide the reasoning process. Experts can express their knowledge at the expertise level, guided by a supervision-oriented description language (YAKL, see chapters 2 and 3).

PEGASE+ accepts two types of complementary declarative descriptions: structural frame-based and rule-oriented. Structural descriptions are used for functionalities, operators, data and arguments, while rules are used for criteria expression.

Structural Descriptions

According to the model, PEGASE+ manages the concepts of Supervision Operators, Arguments, Functionalities, Requests, and Domain Objects. These concepts are implemented by structured objects or *frames*.

The frame formalism (as introduced by Marvin Minsky) is used to model both the knowledge base types and the fact base objects. Frames are useful to represent structured objects and objects interrelations. They are often used to complement the expressive power of production rules. Moreover, inheritance may be used to share relationships and properties.

Frames are complex data structures that represent typical situations or objects, which are organized according to three levels:

1. The *frame level* corresponds to the entities being described. It acts as the type of such entities.

2. The *attribute level* represents the most interesting characteristics of this entity, with respect to the problem to solve. Attributes are called *slots* and correspond to predefined roles in the situation described, or to sub-parts of composed objects, etc. The existence of a slot, even if it is empty, is an indication that can be used by the AI reasoning process. Slots may refer to classical data values such as integers, arrays, etc., or to another frame instance.

3. At the *slot level* each slot is described by sub-attributes or *facets*; facets usually belong to predefined categories. A particular category of facets are *daemons*, *i.e.*, reflex procedures called by the system without user intervention.

In our frame system four categories of facets are predefined for a slot. A facet of each category is present in any slot although it may be empty (i.e., undefined).

The first two facets simply carry values which must be of the type of the slot value: they are *value* (the current value) and *default value*. The third *domain* facet represents the range of definition for the values of the slot. At this time, slot ranges may be of three types: interval, collection, and specialization. Intervals describe sets of possible values ranging from a minimum value to a maximum one—this supposes an ordering relation on the value type. Intervals are usually numerical. Collections extensively provide all possible values in the domain, and generally no order is assumed. Specializations specify a refined root class for the value of the slot: any instance of the root class or of one of its subclasses is acceptable.

Of course the value and default facets must fall within the range of the slot. The corresponding check is performed when setting the value and the default value of the slot.

The last facet is an *if_needed* daemon. This daemon is called when the value of the slot is required by the reasoning process and the *value facet* is empty. The corresponding function must have been previously defined by the expert. It computes the value of the slot based on the current information available. If there is no *if_needed* daemon, the default value facet is returned if it is not empty; otherwise an error is raised.

Decision Criteria Module

The criteria are implemented by specialised rule bases which are local to operators. PEGASE+ incorporates a sub-module implementing a rule engine to process these rules.

In the proposed rule module a rule can be active or not. If it is active, its actions are executed *if and only if* its conditions are all true (which means that there is an implicit *and* between all the conditions in premise). Several instantiations (combinations of objects that can instantiate the free variables) imply that the actions are performed as many times as instantiations.

The rule engine that runs the rules is currently a simple forward chaining engine. Running a rule base means running each rule in it which is executable. The conditions in the rule premise are tested sequentially in the order given by the expert, and the first active one with a true premise is executed, then the rule is deactivated (*i.e.* removed from the list of rules to check), afterwards, the rule engine tries all the other rules, and so on until the list is exhausted.

In PEGASE+ five types of criteria are accepted. They correspond to five types of rulkes. To choose between different alternatives the engine uses *choice criteria*, to tune program before execution it uses *initialisation criteria*, to diagnose the quality of the results it uses *evaluation criteria* and to repair a bad execution it uses *adjustment* and *repair criteria*. The rules are specialised depending on the type of criteria they belong to, which means that the accepted vocabulary is different in rules from different types (see chapter 3). The rules often use information stored in the fact base in domain objects or data.

4.1.3 The Fact Base

The fact base contains facts (instances of data/arguments and domain objects) either given by the initial user's request,or asked the end-user during the reasonning, or deduced by the system. In the initial request the user provides the system with instances of data which are passed as arguments for the first selected operator.

The data instances are created by the execution of the operators. Data play an important role in program supervision because many decisions are based on the information they provide. This is particularly true if processing is data-driven as in image processing. The instances in the fact base can also contain information on access to real computer data (*e.g.*, a `File` instance contains a path and a basename).

In addition to data arguments of operators, some domain objects may also be used during reasoning as they provide complementary information. Those objects are highly dependent on the application domain and can be modeled by the expert in the same way as data types. Although they may not be manipulated directly by programs, the correct use of programs may refer to them. The domain object may be abstract ones (such as a context of use referring to global conditions of processing) or more concrete (such as a patient object in medical imagery, which stores information about the patient that may influence the processing of his/her images). The rules can access (read/write) the information contained in data and domain objects.

The fact base also contains the state tree representing the current reasoning. This tree holds the history of all modifications on arguments or domain objects, during the execution of the operators. The failures, backtracks, changes in some values, etc. are stored there, whence they are available when repair is necessary. They are connected to the descriptions of arguments and domain objects.

4.1.4 Connection with real Programs

Programs to be executed are autonomous entities, independent from and external to the program supervision system. They may run on specific machines or under specific operating systems.

If programs are modular entities of a local software library, their execution is straightforward: the programs are run with current data and adequate parameter values and their final status is returned. In case of programs arranged within a stand-alone application, sophisticated communication and synchronisation with programs must be integrated in execution phase, following specific protocols. The same goes for distant programs that can be reached only via the net. Further, a program supervision system can be a distributed system involving information exchange between its different components and process synchronisation. For all these reasons, we have started to develop a communication manager that ensures communications within the program supervision system. Components communicate with each other thanks to messages that are routed by the communication manager, using a protocole specific to each couple of components. In case the programs to supervise are autonomous entities that pertain to a stand-alone application, the dialogue between the engine, the knowledge base and the programs is handled via a proxy mechanism.

This communication manager will evolve in the near future.

4.2 The PEGASE+ PS Engine

The PEGASE+ global inference engine uses a hierarchical planning technique for its construction phase. A program supervision hierarchical planner works on a hierarchy representing abstraction levels between operators (abstract/composite and real/primitive ones). A hierarchical planner first constructs an abstract plan, then refines it for a particular situation. Hierarchical planning is preferred in program supervision, mainly because the expert often thinks in terms of abstract plan schemes. The relations between the abstraction levels denote mainly alternative (choices) or sequential decompositions. A plan step corresponds to the execution (or decomposition) of an operator in the hierarchy with a particular parameter instantiation.

Given a user's request and a knowledge base, the engine develops an execution plan, which correspond to the successive expansions of composite operators and execution of primitive ones. In fact it develops several tentative plans, some of them may be aborted, due to problems detected during reasoning. The solution plan, if it exists, is produced as a result of program supervision, as well as the output data.

4.2.1 Main Loop

We are going to detail the main loop of the engine. It takes as input a user's request and follows the algorithm sketched in figure 4.2.

In this figure the pieces of knowledge from the knowledge base that are used at each step are written in red. It should be noted that the order in which these pieces of knowledge are used is sometimes fairly different than the order of their declaration in YAKL.

The rest of this section describes each phase of the reasoning process (as presented in section 2.2) in details for PEGASE+. As shown in the figure, the phases (indicated in dotted rectangles) of construction, execution, evaluation, and repair are interleaved, because they closely depend from one another: the results in the one step of construction may influence the next other step, and a backtrack due to the necessity of repairing implies reconsidering the current constructed plan.

4.2.2 Identification Phase

The first phase is a preliminary *identification* phase. In PEGASE+ the identification phase performs a matching between the supervision operators in the current knowledge base and a user's request. It searches the whole knowledge base for (usually composite) operator(s) which may solve the user's problem, on the user's data. The match is based on the functionalities given to some operators by the expert and on the input data (their types and number, see figure 4.3). The best match is the operator whose functionality corresponds to the one requested by the user, and which has the same number of input data, belonging to the same types (or a super-type) as the data provided to the system by the user. One matching operator is chosen as root node for the next phases.

This implies that functionalities and arguments types have been properly defined by the expert. If several operators match, the first found is taken (in depth-first search from the root of the operator hierarchy).

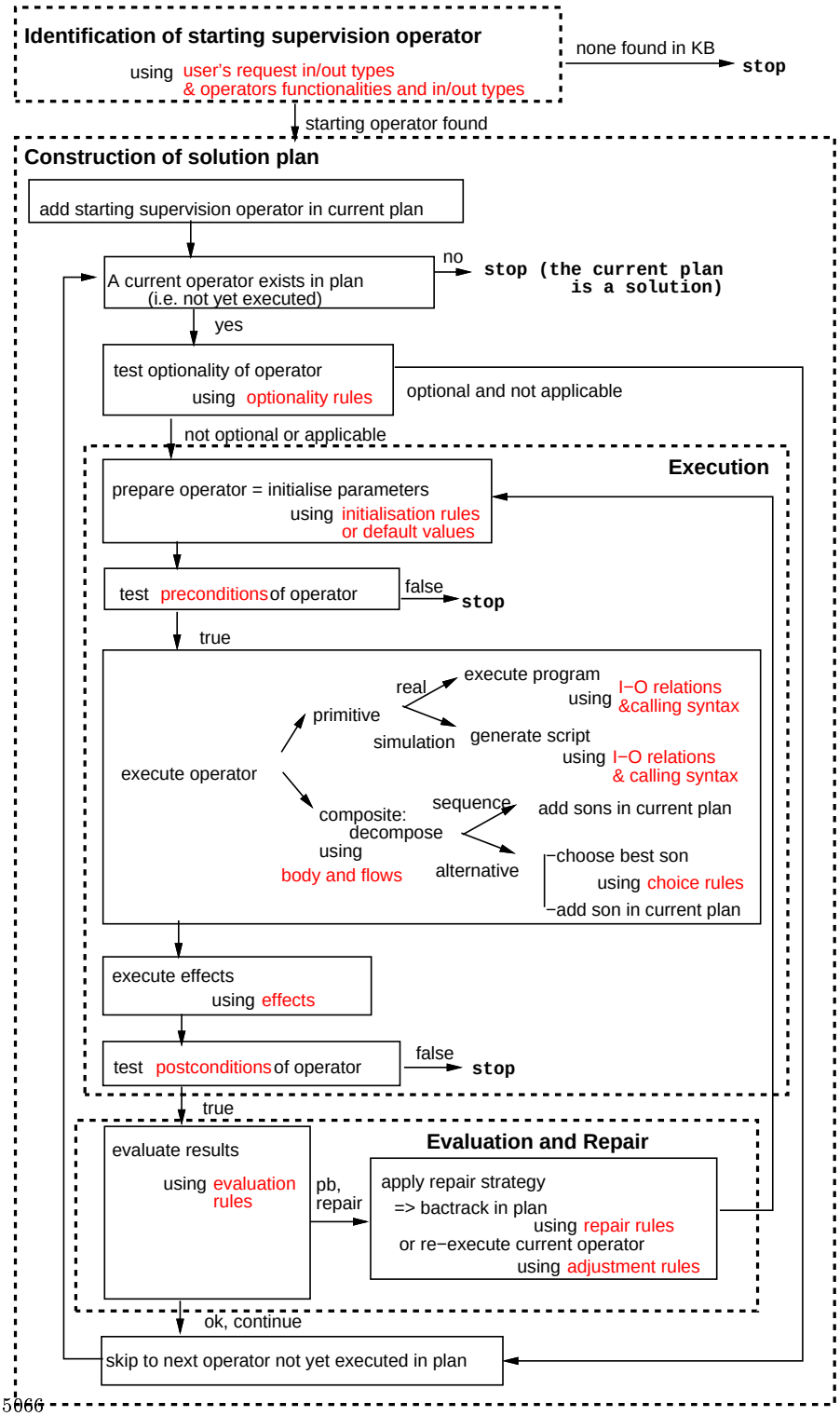


Figure 4.2: Schetchy Algorithm of PEGASE+ Engine Reasonning

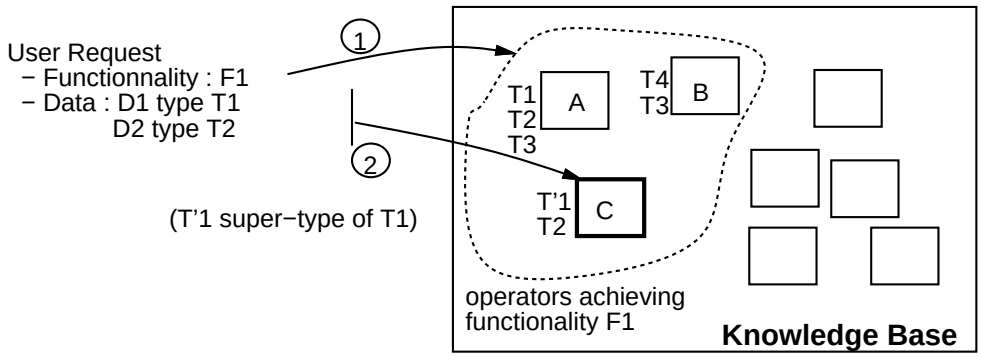


Figure 4.3: PEGASE+ first restricts the search to operators with the same functionality as the one indicated in the user's request, then it selects among those the one(s) that have the same number of arguments, with types compatible with the data provided in the request (here C).

4.2.3 Construction Phase

The construction phase then selects and assemble operators in order to increment the current partial plan. This phase performs a depth-first search in an operator hierarchy, starting from the previously selected root operator (in identification phase). One single hierarchy is chosen in the identification phase and the engine goes through this unique hierarchy, which must therefore be complete, to find the best path.

This phase exploits this hierarchy composed of composite operators, of their descendant and of their attached criteria (used for choices in particular). During this phase, composite operators are expanded, their sub-operators are scheduled to be processed, and the plan is refined. To refine a plan P_0 means to transform it by going one step downwards in the abstraction levels of the hierarchy. This refinement operation leads to a derived plan, where each operator is either an operator of P_0 or an operator derived from an operator of P_0 by expansion of composite operators.

Only the current operator in the rank-ordered plan of PEGASE+ is expanded at a time. First it is initialized (for parameter setting) using the initialization criteria. The expansion then proceed as follows.

The children of a composite operator with a sequential decomposition type are added to the plan, in the order given by the expert. If there is an optional sub-operator, it is also added to the plan along with its applicability condition. The condition will be tested dynamically just before the execution or the recursive expansion of the optional sub-operator. If the condition is not met the operator will be skipped.

For a current composite operator with a specialization/alternative decomposition type, the engine uses the attached choice criteria to select, among all the available sub-operators,

the one which is the most suitable, according to the data descriptions and the characteristics of the operators. The syntax of choice rules allows the use of specific functions, such as *use_ operator_ (of_ characteristic)* and *refuse_ operator_ (of_ characteristic)*. Their effect is to increase (resp. decrease) the degree of interest associated to the sub-operators. The sub-operator with the highest degree will be chosen (in case of equality, the first one in the body list is preferred). The functions that take into account characterisics are less rewarding than the others (i.e. *use_ operator_ of_ characteristic x* if it applies to operator op, will increase less its degree of interest than *use_ operator op*). The sub-operator with the highest degree will be chosen. In case of equality, the first one in the body list is preferred. A sub-operator marked as refused will not be considered for the current time (but it may be later if the reasoning comes back to the choice point after bactrackting).

In a partial plan under construction both composite and primitive operators may appear, but in the final solution(s), only primitive ones are allowed.

Figure 4.4 shows a simple example of an operator hierarchy. In this hierarchy there are both primitive (*e.g.,* 2), and composite operators that will be expanded during planning (*e.g.,* 3).

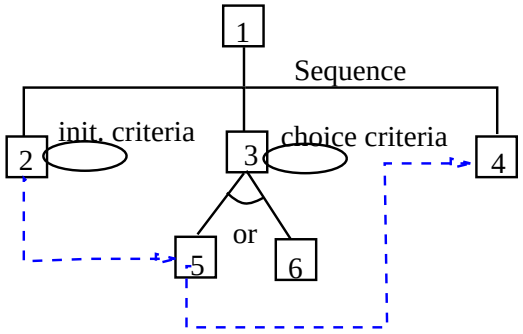


Figure 4.4: Phase of construction in PEGASE+: operator 1 is decomposed into its sub-operators, operator 2 (primitive) is executed, after application of its attached initialization criteria, then the choice between operators 4 and 5 is performed using the choice criteria, etc. The (blue) dashed arrows represent the first plan generated by the engine [2, 5, 4].

When all the sub-operators are expanded, and recursively decomposed or executed, the "effects" of the composite father operator are executed.

4.2.4 Execution Phase

If the current operator to treat is a primitive one, the execution phase runs (directly or via a communication protocol) the program associated with it.

It starts by setting the values of their input arguments: the data and parameters indicated in the calling syntax are replaced by their actual values. For parameter initialization, the

engine first uses the initialization rules, then the default values for setting parameter values. If some parameters still have no values, the execution is aborted, and the engine stops.

Concerning input data, it uses the data flows: distribution -coming from the father operator- or flow -coming from brothers. The initial values of data are given by the request, and passed trough the flows. Some data may come from the output data of other operators, that is why planning is interleaved with execution and repair phases. Each planning step may depend on information that is only available after the execution of previous programs in the plan and the exact form of the plan typically depends on information that is available at execution time only.

Afterwrds, the preconditions are tested to check if the operator should apply in the current state, if not the engine stops. The execution itself calls the program [1]. Depending on the chosen mode -real or simulation- it either concretely executes the call or only prints a script, without real execution. The final objective of program supervision is usually the actual execution of programs, eventhough the simulation mode may be useful, for testing purposes for instance. After execution, the postconditions are tested. If they do not hold in the resulting state the engine stops. Finally, the effects of the operator are executed.

4.2.5 Evaluation Phase

The next phase is the evaluation (or assessment) of the operator results (output data). For a composite operator, it comes after the processing of all its sub-operators, for a primitive one its comes just after its execution. Evaluation fires the evaluation rules attached to the current operator. Their goal is to label the operator as either "correct" or "to be repaired". These rules may be either completely automatic (if the expert knows a quantitative way of evaluating the results) or interactive (if the expert indicates a dialogue with the end-user about the features of the results). Dialogs with the end-user are predefined by the expert by means of `ask-user` functions in the evaluation rules. The evaluation then operates automatically for quantitative evaluation and is performed manually (by the user) for qualitative evaluation. Sometimes one may need to use programs to evaluate other programs. PEGASE+ does not provide any help to construct these evaluation programs, but thanks to evaluation criteria a mechanism to use them when they do exist. Moreover a module for curves evaluation is provided to the experts, allowing automation of the evaluations on results, if they are curves.

In fact PEGASE+ works on results *descriptions* not on real results. If real results are stored in files or variables for instance, output data descriptions can collect and store information about them (using I-O relations or rules, see chapter 2).

The evaluation rules express judgement (using `assess_..` functions) and thus modify the "`judgement`" attribute of operators and arguments.

Evaluation in primitive operators are usually difficult, they are easier in composite and more abstract ones.

[1] In the future it will be possible to specify the kind of execution (local or remote) see section 4.1.4.

4.2.6 Repair Phase

Finally, if the assessment on results is negative (*i.e.* if the current operator has been labeled "to be repaired", the repair phase calls the local repair and/or adjustment criteria (rule bases) to decide how to repair it in order to improve the results. This could command backtracks to previous steps in the plan, depending on the type of decisions that can be reconsidered. In PEGASE+ there are two main repair actions: re-execution, which takes the current plan and tries to find better values for tunable input parameters; and replanning, which tries to find another plan. The repair method is provided by the expert, in repair rules, when he/she models the program or the complex processing.

Several functions can be used in repair rules to express repair strategies (see also section 7.3.7). They work on an already executed plan that needs some repair. The plan to repair is composed of a sequence of steps, corresponding to the successive executions/expansions of operators. The expert's repair strategy can be divided into several steps: problem propagation (if the current operator cannot repair the problem locally), plan revision (if the structure of the plan itself and the choices made are questionable), or local adaptation (if the current operator can repair the problem locally by re-executing with new parameter values). Problem propagation and plan revision lead to moves in the state tree. Each repair strategy should end either on a local adaptation, *i.e.* after moving in the state tree, the current state corresponds to an operator that has the knowledge to repair the problem by re-executing itself, or on the execution of a new operator (*i.e.* not previously in the plan) which will produce satisfactory results.

E.g., the expert can express that the bad evaluation information has to be transmitted to a sub-operator or to the father operator in the hierarchy, or to any operator previously applied. The control then switches to the target operator, whose repair strategy is then activated, based on the diagnosis that has been sent to it. The problem must eventually attain the offending operator, which can repair it by re-executing itself with adjusted parameter values.

Here are the functions managed by PEGASE+ for these different steps.
For problem propagation:

- *send_ up* transmits a problem to a father operator in the operator hierarchy. In the example of figure 4.4, this function allows, if the evaluation after execution of step 4 is not correct, to delegate the repair to its father (1). The more global knowledge associated with 1 will be used to find a better solution. For example, the repair rules of 1 may express that, since the most sensitive sub-operator is 2, the problem must be transmitted to it. And consequently, this operator will trigger its own repair knowledge.

- *send_ down(child1)* transmits a problem to a child (named child1). A father is able to delegate the repair to one of its children. For example, in the previous case, 1 transmits the problem to 2, by the command *send_ down(2)*. If we suppose that when 2 receives a problem (directly by the evaluation mechanism or by its father) its repair

knowledge implies a *re_ execute* action, a new instantiation of this operator will be re-executed after adjustment of its parameters.

- *send_ operator(op_ name)* transmits a problem to any (previously involved in the plan) operator.

For plan revision:

- Return to the last (temporal order) choice: *back_ choice* makes the planner return in the state before the last choice, in order to try another solution. In the example of Figure 4.4, if the evaluation, after execution of step 4 is not sufficient, *back_ choice* will force the planner to backtrack to a less refined plan, where operator 3 is not yet refined.

For local adaptation:

- Adjust the current operator: *re_ execute* triggers the adjustment criteria (if they exist) on the current operator. Ajustement criteria compute a new value for each questionable parameter, not all parameters must change it depends on the problem detected in evaluation criteria. Parameters that are not changed by the applied adjustment rules keep the same value as in the previous execution. Values may be completely changed by assignments or they may be increased or decreased starting from the current value. In the last case, the parameter *must* have a range, because the increase/decrease process is based on the limits of this range.

  Parameters are most often numerical ones, and their value may be increased or decreased, according to their adjustment method (which may be changed in rules). Three of the most common numerical adjustment methods are predefined in YAKL language: *i.e.*, taking a certain percentage of the original value, adding a constant to the original value, and splitting the region of possible parameter values into two distinct regions, and adapting the parameter to the middle of one of these regions (by dichotomy). But the expert may define new ones, which must respect the expected signature: `T expert_method(T,T,T)` where `T` is the type of the current parameter. For numerical parameters `T` is `int` or `double`). When the engine calls the method, the first T argument will be filled by the current value, the second one by the increment/decrement step and the last one by the minimum or maximum of the interval (only if the range is an interval). It should be noted that the adjustment method must return a value that will be added (resp. substracted) from the current one and *not* the resulting final value. Indeed the final value is equal to the current value plus (resp. minus) the result of the adjustment method.

  Increasing (resp. decreasing) symbolic or string values, as well as numerical values that have an enumeration as range, is fixed and predefined in the PEGASE+ engine: to increase means to take the next value in the range, and the reverse way for decrease.

  For other types of parameter values, since there is no admitted semantics on what does increase/decrease such values mean, the engine does not use the adjustment methods. The only way to adjust such parameters is to assign them a new value.

Functions *send_up* and *send_down* have a sense only for hierarchical planners, which is the case for PEGASE+.

After an evaluation/reparation, a new line of execution starts (with new parameter values).

4.2.7 Error Messages

The following messages may happen during the execution of a KBS build using PEGASE+:

"<Engine>: No adequation for initial request: " IDENT
The types and names of in/out data of the current request (named IDENT) do not match with the ones of any operator in the knowledge base. Check types and names in your operators and functionality/request.
"<Engine>: No scheme for initial request " IDENT
No entry point has been found in the knowledge base operator hierarchy the current request (named IDENT) i.e. no operator matches the required functionality. You may have forgotten an "Achieved by.." line in a Functionality definition and/or a "Functionality .." line in an Operator definition.
"<Engine>: "No request - STOP"
No request has been defined in the `.yakl` files.
"<Engine>: "Several operators match the funtionality data types of request, first is chosen; Possible operators are: " list of IDENT
In the case when several operators (whose names aregiven by list of IDENT) in the knowledge base match the funtionality required by the current request, one is chosen randomly.
"<Engine>: "No Program Supervision base, stop"
The C++ code generation (during parsing) did not produce a correct main program, and no knowledge base has been defined. This should normally never occur if parsing was successful.
"<Engine>: "Invalid debug level, minimum assumed" or *"<Engine>: "Invalid request number, first assumed"*
These messages correspond to an incorrect call to the knowledge-based system executable code i.e. a command line with arguments out of bounds (e.g. a negative debug level).
"<Engine>: "Stop on exception " IDENT or *"<Engine>: "Unexpected system exception"*
These messages may occur when an exeception (a PEGASE+ one or a system one) occurred during execution. The TEXT is an information about the PEGASE+ function which raised the exception. This denotes a serious problem.
"<Engine>: "Repair unable to find backchoice operator" or *"<Engine>: "Repair unable to find previous operator:* IDENT
These messages may occur during the repair process, when the engine is unable to backtrack to a given point in the previous reasonning states. Maybe the operator you chose for backtracking does not exist in the base?.. or has not been executed in the past of the current state?

Part II

User's Manual

Chapter 5

Yakl Grammar

The YAKL language has been developed to allow experts to describe all the different types of knowledge involved in program supervision, independently of any application domain, of any program library, or of the implementation language of the knowledge-based system (in our case C++). YAKL is used both as a common storage format for knowledge-bases and as a human readable format for writing and consulting knowledge bases. YAKL descriptions can be checked for consistency, and eventually translated into operational code.

The language offers two types of declarative descriptions: structured frame-based and rule-oriented. The frame formalism has been chosen to represent functionalities, operators, data and arguments, while inference rules are used to express the various criteria that are applied during the reasoning process. All the rules follow the same general syntax, but each kind may have its own vocabulary.

This chapter describes in detail the BNF (Bacchus Naur Form) grammar rules of YAKL, with the following syntactical conventions:

- A grammar rule is represented by:

  ```
  one-non-terminal : sequence of terminals or non-terminals (maybe empty)
  ```

- Alternatives are indicated by a pipe (|);
- YAKL key-words are indicated in bold face: ex: **Operator**;
- Other terminals of the grammar are indicated in upper-case:
  - IDENT denotes a symbol, beginning by a letter or a digit, and that may include _ or digits in the following; *Warning:* identifiers are interpreted either as names of arguments of current operator or as domain object names or as variable names. As a consequence, be careful **not to use** a variable that has the same name as

an argument or a domain object (nor of a keyword of the programming language, like `long` in C++).

- COMPOSEDIDENT denotes a sequence such as IDENT.IDENT.IDENT... which represents access to sub-sub...-attributes (or methods)of frames, *i.e.* data, parameters, or domain objects; For instance, the syntax <*frame_name*>.<*attribute_name*> allows in a rule to access to the value of the attribute named *attribute_name*, of the frame named *frame_name*. The frame may be *e.g.* an operator and the attribute a data or a parameter, or the frame may be a domain object and the attribute one of its own. A call to a *method* is also possible in composed identificators (notation: IDENT(...), if IDENT is the name of a method of the frame or of the sub-attributes just before it in COMPOSEDIDENT), as well as a reference to an element of a collection (notation: IDENT(INTEGER), if IDENT is the name of an attribute of type Set, with at least the value of INTEGER elements).
- FLOAT denotes a float number with decimal point;
- INTEGER denotes an integer number;
- STRING denotes any sequence or characters between two ";
- CODE denotes any code source piece in C++;
- SYNTAX denotes a calling syntax for an operator, it may include options (it corresponds to the line you type in the shell for example).
- LIST stands for a sequence of numbers or symbols, separated by blanks and enclosed into brackets ([...]).

- Comments and marginal notes explain some points of this grammar.

All along this chapter YAKL source pieces are provided to illustrate the rules when possible.

5.1 Start

```
pskb : kb_desc                                   Global KB description, .kb file apart
     | import_list dcl_list                      or 'regular' .yakl file
          arg_type_def arg_instance_def
          obj_type_def obj_instance_def
          operator_or_functionality_def request_def

operator_or_functionality_def : operator_def
     | functionality_def
```

A program supervision knowledge base description in YAKL is composed of:

- Either a description of the knowledge base as a whole (`kb_desc`) in a separate file (with a `.kb` extension), see next section (5.2) for details.
- or a 'regular' `.yakl` file (i.e. containing description of operators, rules, etc.) composed of
  - a set of imported file names (`import_list`).
  - a set of declarations of functions or global variables, necessary for some code written by the expert in separate C++ files (`dcl_list`). These are typically declarations of signatures of functions called into CODE parts. This is similar to the `extern` declarations in C++.
  - definitions of argument types and instances (`arg_type_def` and `arg_instance_def`).
  - definitions of domain object types and instances (`obj_type_def` and `obj_instance_def`).
  - definitions of operators or of functionalities in any order (`operator_or_functionality_def`).
  - definitions of requests (`request_def`).

5.2 Global KB Description

The knowledge base contents is defined only once, in a separate `.kb` file. The root node(s) is(are) the one(s) at the top of the (preferably unique for PEGASE) hierarchy(ies) of operators described in the knowledge base. Indicating the absolute path (**KB Path**, where all `.yakl` files are assumed to be) is optional, by default current directory is assumed. Optional code file names indicate where is some additional code, written by the expert (in C++), to complement the knowledge expressed in YAKL; this code may concern methods or functions that the expert wants to call, e.g. in rules or in pieces of CODE.

kb_desc : **Kb { name** IDENT
Complete Name strings — *strings is a sequence of STRINGs, separated by blanks*
Authors strings
path
Version FLOAT
Root Node ident_list
codefiles
List of Files ident_list
codefiles **}**

path : — *May be empty*
| **KB Path** strings

codefiles : — *May be empty*
| **Code Files** ident_list — *(C++) code files for methods or functions*

Example

Kb { **name** star
Complete Name "Star detection"
Authors "L. and S."
Kb Path "Z:/Supervision/Example/"
Version 0.9
Root Node detect_star
List of Files star1 star2
}

5.3 Regular Files

The `.yakl` files constitute the "flesh" of the knowledge base, they contain the descriptions of all operators, instances of objects, rules, etc. that all together are the knowledge to be used by the engine. You may have several of these files, but try to keep a convenient size (not one file for each single operator and not one unique file for all operators!).

5.3.1 Imported files

The **Import** lines indicate the names of other KB files, needed for the description of the current file contents. They contain YAKL source. The list of imported files should be given without the `.yakl` extension. For instance, if an operator needs previously defined types or operators, you must *import* the files where these types or operators have been defined, in a similar way to the #include mechanism in C. If a file imports a lot of other file several **Import** lines may be necessary.

import_list : *May be empty*
| import_list import_line
import_line : **Import** ident_list *Ends with end of line*

Example

Import declaration-file simple-ops-file composite-ops-file

5.3.2 External Declarations

The purpose of this section is to declare function signatures (or variables) that are needed inside the current file (usually in a CODE part). These functions must be defined in some code file, Declarations are done one per line, each line starting by the keyword **Extern**.

decl_list : *May be empty*
| decl_list decl_line
decl_line : **Extern** CODE *a correct declaration, one per line*

NB: The contents of the declaration is not parsed and must be correct in the target programming language, otherwise problems will arise when compiling the base. Note also that methods (of argument types, for instance, see next section) need not to be declared this way, because they are part of a type and thus declared by simply importing the `.yakl` *file that contains the corresponding type.*

Example

Extern double compute_value(int, const String &)
Extern int global_var

5.3.3 Definition of Argument Types and Instances

The expert can define the necessary types of the arguments of the target programs, if they are not predefined. YAKL provides a standard hierarchy of predefined types for data (with predefined attributes and methods), either simple: *Integer, Float, String, Symbol* (with predefined attributes), or structured: *Image, File* (with additional attributes and methods the expert may need). *Image* is currently defined as a sub-class of *File*, so both have the following attributes (of type *String*): basename, path and extension. They also have one predefined method: *get_filename()* which returns the complete name of the file. The *Image* type has two more slots: x_size and y_size of type *Integer*.

Argument Types

The structured type hierarchy can be extended or overridden by the expert to describe any complex data structures involved in the target application domain. New expert-defined types may have methods. For each slot of a data type, the expert can specify a range of possible values and a default value. The end-user or the engine during reasoning may in turn provide a specific value.

arg_type_def : *May be empty*
| arg_type_list
arg_type_list : arg_type
| arg_type_list arg_type
arg_type : **Argument Type {** **name** IDENT comments
arg_subtype_of attributes methods **}**

arg_subtype_of : *May be empty*
| **Subtype Of** IDENT

attributes :
| **Attributes** attribute_list
attribute_list : attribute
| attribute_list attribute

methods :
| **Methods** method_list
method_list : method_signature
| method_list method_signature

method_signature : IDENT IDENT (ident_list)
comments :
| **comment** strings
strings : STRING
| strings STRING

*NB: Each new type specifies its list of authorised method names. Method bodies should be written in separate C++ files (***Code Files***, see section 5.2)). A method signature corresponds to the type of its arguments and its return type. Note that no verification is done by the parser on method arguments (nor the types neither the number of arguments are verified).*

Attribute Declaration There are two forms for attribute declarations, so that experts can choose the one they prefer. The first one – which may look "computer-oriented" to some experts– is:

attribute : IDENT	*1st IDENT is the type name*
name IDENT	*2nd IDENT is the attribute name.*
comments attribute_info	
| **Forward** IDENT	*1st IDENT is a yet to be defined type name*
name IDENT	*2nd IDENT is the attribute name.*
| **Override** IDENT **name**	
IDENT	*2nd IDENT is a redefined attribute name.*
comments if_needed	
| **Set of** IDENT **name** IDENT	*a collection*
comments attribute_info	
| **Set of Forward** IDENT **name** IDENT	
comments attribute_info	

*NB: **Forward** indicates a forward reference, it is used when one cannot avoid referencing a type that has not been defined yet, typically in the case of two types, that reference each others (an attribute of the first type is of the second type and the reverse). Do not over use this feature! **Override** is used to override the type definition of a super-class. By overriding an attribute of a super class, the expert may refine its if_needed daemon (and* **only** *this*

facet in the current version).
Set of *allows to define attribute types containing collections of objects of the same type (first IDENT). Note that sets cannot be overriden.*

Example

Attributes
Integer name x
Float name y
File name data_file

There exists a second version of attribute declaration, equivalent to the first one – keywords have the same meaning– which may seem more natural to some experts:

attribute : IDENT — *1st IDENT is the attribute name*
a/an IDENT — *2nd IDENT is the type name.*
comments attribute_info
| **Override** IDENT
a/an IDENT
comments attribute_info
| IDENT **a/an**
Forward IDENT — *2nd IDENT is a yet to be defined type name.*
comments attribute_info
| IDENT **a/an Set of** IDENT — *a collection*
comments attribute_info
| IDENT **a/an Set of Forward** IDENT
comments attribute_info

Example

Attributes
x **an Integer**
y **a Float**
data_file **a File**

Attribute Information In both cases the information attached to an attribute is the same:

attribute_info : default range if_needed

default :
| **default** value

if_needed :
| **calculation** CODE
| **calculation** IDENT
| **calculation** item_file

*NB: if_needed (**calculation**) is a so-called «daemon», i.e. a way to compute automatically a value, if the value is necessary during the reasoning but not yet available. It could be a small piece of code, a call to a function (defined in a code file and declared in the declaration part of the current file), or an access in a file, using the **item** notation (see page 64).*

value : IDENT
| FLOAT
| INTEGER
| STRING
| | value_set | *for a collection*
| **nil** *for an empty collection*
| **{** CODE **}**
range :
| **range [** interval **]**
| **range [** value_set **]**
interval : FLOAT , FLOAT
| INTEGER , INTEGER
value_set : ident_list
| float_list
| int_list
| string_list
ident_list : IDENT
| ident_list IDENT
float_list : FLOAT
| float_list FLOAT
int_list : INTEGER
| int_list INTEGER
string_list: STRING
| string_list STRING

NB: The range of an attributes describe the possible values of this attribute, for numbers it corresponds to the usual [min, max] interval domains, for symbols (strings) is displays all the acceptable values. Collections can be assigned a default value as a whole: either a set of values (that must be of the right type) or the empty value (nil).

Example

```
Argument Type { name GData
        comment  "general, for all types of data"
        Subtype Of Image
Attributes
        Symbol name sort
            comment "to distinguish 2 possible kinds of data:
                pixel images or tables of numbers"
            range [ pixels float ]
Methods
        Float mini (), # return the minimum value in the image
        Float maxi () # idem for the maximum
}

Argument Type { name  Histogram
        comment "Image histograms"
Attributes
        Integer name nb_classes # nb of classes in histogram
            calculation {{return 20;}} # Dummy code for the moment
        Integer name nb_empty_classes # nb of classes with no element
Methods
        Bool holes() # return true if there are 'holes' in histogram
            # repartition and computes new min and max
}
```

Argument Instances

Once the types defined, experts can define instances of these arguments, these instances will act as "object values", to be used in request attribute assignement (see section 5.3.5) or to set the values of attributes of other instances.

```
arg_instance_def :
        | arg_instance_list
arg_instance_list : arg_instance
        | arg_instance_list arg_instance
arg_instance : Argument Instance{  IDENT     1st IDENT is the argument type name
               name : IDENT comments         2nd IDENT is the instance name.
               attribute_assignments }

attribute_assignments :
        | Attributes attribute_assig_list
```

attribute_assig_list : attribute_assig
| attribute_assig_list attribute_assig
attribute_assig : IDENT := value *IDENT is an attribute name*
| IDENT := { attribute_assig_list } *Inline description of sub-parts*

*NB: When an object is composed of sub-part objects, the expert may either define sub-objects apart (***before** *the composite one, in the same file or in an imported one), name them and use their names as values to set attributes of the global object, or describe the attributes of the sub-objects directly, in nested curly braces. The first option should be prefered when sub-objects have a lot of attributes to fill, while the second option is easier when sub-objects have only a few attributes with interesting values.*

Example

```
Argument Instance {
        Image name my_image # instance of predefined type
Attributes
        path := "/u1/Star/"
        basename := "sky1"
        extension := ".jpg"
}

Argument Instance {
        GData name d1 # instance of expert-defined type
        comment "Simple data for test"
Attributes
        path := "/u/Example/" # attribute inherited from File
        basename := "data1" # attribute inherited from File
}
```

5.3.4 Definition of Domain Types and Objects

In the same way, experts can define domain types and objects, if they are necessary to the program supervision reasoning. They will not be used as arguments of operators but they may appear for instance in rules to guide the reasoning process.

Domain Types

obj_type_def :
| obj_type_list
obj_type_list : obj_type
| obj_type_list obj_type

obj_type : **Object Type { name** IDENT comments
obj_subtype_of attributes **}**

obj_subtype_of :
| **Subtype Of** IDENT

Example

```
Object Type {
      name Context
      comment "global context of use of the system"
Attributes
       Symbol name automatic_mode
            default no
            range [yes no ]
      Symbol name systematic_display
            range [yes no ]
}
```

Domain Objects

obj_instance_def :
| obj_instance_list
obj_instance_list : obj_instance
| obj_instance_list obj_instance
obj_instance : **Object Instance {** IDENT *1st IDENT is the argument type name*
name IDENT *2nd IDENT is the instance name.*
comments attribute_assignments **}**

Example

```
Object Instance {
      Context name context1
Attributes
      systematic_display := yes
}
```

5.3.5 Definition of Functionalities and Requests

Functionalities

Functionalities are abstractions of processing actions, they may correspond to (be achieved by) several concrete operators. Their descriptions are similar to operator ones, as far as data and parameters are concerned.

functionality_def : **Functionality { name** IDENT comments
achieved input_data parameters output_data **}**
achieved : *May be temporarily empty*
| **Achieved by** ident_list *List of operators*

input_data :
| **Input Data** input_data_list
output_data :
| **Output Data** output_data_list
input_data_list : input_datum
| input_data_list input_datum
input_datum : type **name** IDENT comments default
parameters :
| **Input Parameters** parameter_list
parameter_list : parameter
| parameter_list parameter
parameter : type **name** IDENT comments default range

output_data_list : output_datum
| output_data_list output_datum
output_datum : type **name** IDENT comments default io_relations

type : IDENT *IDENT is a type name, must have been already defined.*
| **nil** *Nil stands for a default type.*

io_relations :
| **I-O relation** relation_list
relation_list : relation
| relation_list , relation
relation : COMPOSEDIDENT := exp *:= means assignment*
| IDENT := exp (of sub..sub) slots) of current output data,
| IDENT = IDENT = *means complete copy of input data to output data*

*NB: **I-O relations** are used to express relationships between data descriptions of an input and an output data (e.g., files may share the same directory, basename, etc.). Assignment of attributes (:=) implies sharing of values, while copy of data as a whole (=) means that,*

after the copy, the arguments will behave independently. I-O relations are more useful in operator descriptions.

Example

```
Functionality {
        name star_detection
Achieved by detect_stars
Input Data
        Image name im
        comment "raw image"
Output Data
        File name objects
        comment "file containing list of detected objects"
}
```

Requests

Requests come from users; they are instances of functionalities, they correspond to calling an abstract processing on some concrete data, without willing to know which concrete operator will achieve the processing.

```
request_def :
        | request_list
request_list : request
        | request_list request
request : Request {  IDENT                    1st IDENT is the name of a functionality,
        name : IDENT comments                 2nd IDENT is the name of the request
             attribute_assignments }
```

Example

```
Request {
        star_detection name detection1
        comment "Higher level abstract functionality"
Attributes
        im := my_image
}
```

5.3.6 Definition of Supervision Operators

Head of Definition

operator_def :
| operator_list

operator_list : operator
| operator_list operator

operator : **Primitive Operator** { operator_primitive }
| **Composite Operator** { operator_composite }
| **Local Operator** { operator_local }

operator_composite : operator_common body_plan
operator_primitive : operator_common call
operator_local : operator_common **Call** rule_list

NB: Local operators are used for simple processing steps that do not necessarily correspond to a real executable code, their calling syntax equivalent is simply a list of rules (instead of the calling syntax corresponding to an executable external code in primitive operators).

Common Part

operator_common : **name** IDENT **Authors** strings
comments functionality characteristics
input_data parameters output_data
preconditions postconditions effects
initialisation assessment repair adjustment

NB: Assessment criteria describe a precise means for result evaluation for each operator.

functionality : *May be empty*
| **Functionality** IDENT

characteristics : *May be empty*
| **Characteristics** ident_list *just a list of symbols.*

preconditions :
| **Preconditions** condition_list *true before execution*
postconditions :
| **Postconditions** condition_list *true after execution*

effects :
| **Effects** effect_list

effect_list : effect
| effect_list , effect
effect : IDENT := exp
| COMPOSEDIDENT := exp
| { CODE }

NB: **Effects** *express the effects (consequences) of an operator execution on its output arguments (**not** on the domain objects). They are thus executed after the operator execution.*

condition_list : condition
| condition_list , condition
condition : **valid** IDENT
| IDENT compar exp
| COMPOSEDIDENT compar exp

*NB: **valid** is a predefined method, that checks if the argument really exists (e..g., in memory for a file). It is predefined for* YAKL *types such as* `Image`, `File`, *etc. It has obviously no sense for scalar types (*`Integer`, `Float`, `String`*)*

compar : <> | == | < | > | <= | >=

exp : (exp)	
| value	
| COMPOSEDIDENT	
| **max** (exp exp)	*maximum of 2 expressions*
| **min** (exp exp)	*minimum of 2 expressions*
| exp + exp	*sum of 2 (numerical) expressions*
| exp **-** exp	*difference*
| exp ***** exp	*product*
| exp / exp	*division*
| **item** INTEGER cast **in** file item_ref	*access to a file*
file : STRING	*name of the file, as a string*
| IDENT	*name of the file*
| COMPOSEDIDENT	*name of the file, as result of a call*
item_ref :	*Optional*
| **line** IDENT	*significant word in line to search*
| **line** STRING	*significant word in line to search*
| **line** COMPOSEDIDENT	*significant word, result of a call*
cast : (String)	*extracted string will be converted to a string*
| (Integer)	*extracted string will be converted to an int*
| (Float)	*extracted string will be converted to a double*

*NB: When results are collected in a file they can be accessed by the **item** notation. The place to search for the interesting information is indicated by a number (an INTEGER), which*

is the index of a word in the whole file or in a line. In the corresponding rules, STRING stands for a file name (the file is either named directly or its name may be computed thanks to an attribute value or a method call expressed by COMPOSEDIDENT). The place in the file is indicated by the INTEGER (= word index), and the the line to search may be specified by an optional key-word, which is any significant word appearing in it (item_ref).
Note that in the index computation ***delimiters (space tab ; : =) don't count as words!****.*

initialisation :
| **Initialization criteria** rule_list
assessment :
| **Assessment criteria** rule_list
repair :
| **Repair criteria** rule_list
adjustment :
| **Adjustment criteria** rule_list

Primitive Operators

call : **Call language** language call_list
| **Call language** language call_list type
call_list : a_call
| call_list a_call
a_call : **syntax** SYNTAX **endsyntax**
type :

| **type** real	*concretely execute the actions*
| **type** simulation	*print actions, but do not execute them*
language : **C++**	*commands may be either internal (linked)*
| **Matlab**	*routines in C++ or Matlab routines*
| **shell**	*or external programs called via the shell*

NB: You may have several actions for the same operator. For instance, first remove a file, then execute a program creating this file, that is why you may write several ***syntax*** *lines. All actions share the same language (shell, or C++, or Matlab) and the same* ***type*** *of execution (real or simulation).*
Note that, to be able to call Matlab functions you must use a version of PEGASE+ *that has been specially tuned.*
In the future, further extensions of the calling information will be added for remote programs.

Example

```
Primitive Operator{ name display_pixel_image
        comment "Simple display"
        Authors "A. L."
        Characteristics pixels
        Input Data
                GData name input
        File name value_table
        Preconditions
                input.sort == pixels # only for pixel images
Call
        language shell
        syntax display input.get_filename() value_table.get_filename()
        endsyntax
        # input.get_filename() is a string, the name of the file,
        # it is thus acceptable as an element of a command (cf Appendix 2)
        type real
}

Primitive Operator { name histogram
        comment "Compute image histogram"
        Characteristics float
        Input Data
                GData name input
        Input Parameters
                Float name y_min
                Float name y_max
                Float name step
        Output Data
                Histogram name histo
        Preconditions
                input.sort == float # only for float number tables
...
Call
        language shell
        syntax histo input.get_filename() y_min y_max step
        endsyntax
        type real
}
```

Composite Operators

body_plan : **Body** sub_op_list
body-criteria *for choices or optional sub-operators*
Distribution link_list
flow parameter_flow

*NB: **Distribution** is data flow between a parent operator and its children and **Flow** is data flow among children in a sequence.*
*An operator used as an optional sub-operator in a body **must** have the same number of input data and of output data (extra data are allowed if they are at the end) in the same order and of the same type. When the operator is not executed because the optional criteria does not apply, the input data are directly passed to the output data. This data flow should not be expressed in the data **Flow** part.*

sub_op_list : IDENT | or_decomposition
| IDENT **-** seq_decomposition
| IDENT |
| IDENT **-**
| [IDENT] -
| [IDENT] - seq_decomposition
| IDENT || par_decomposition
body-criteria :
| **Choice criteria** rule_list
| optional_list
optional_list : optional_rule_base
| optional_list optional_rule_base
optional_rule_base : **Optional criteria for** IDENT rule_list

*NB: One has to write one rule base for each optional operator in the body. The IDENT after «**Optional criteria for**» is the optional sub-operator name.*

link_list : link
| link_list link
link : IDENT **.** IDENT / IDENT **.** IDENT
| IDENT **.** IDENT **:=** exp

NB: In distribution or flow 1st IDENT stands for the name of the (sub)operator, and 2nd one for the name of an attribute. It is similar to a formal substitution of attribute name.

or_decomposition : IDENT
| or_decomposition | IDENT

seq_decomposition : IDENT
| **[** IDENT **]**

| seq_decomposition **-** IDENT
| seq_decomposition **-** **[** IDENT **]**

par_decomposition : IDENT
| par_decomposition || IDENT
flow : *Data flow among sons is empty, in a choice*
| **Flow** link_list

parameter_flow :
| **Parameter Flow** link_list

NB: ***Parameter Flow*** *allows parameters to be passed from a parent to its children.* ***Flow*** *and* ***Distribution*** *are kept for data.*

Example

```
Composite Operator { name display_float_datax
        comment "Discretize before display"
        Input Data
                GData name input
....
        Body histogram - display_pixel_image
        Distribution
                display_float_data.input / display_pixel_image.input
                display_float_data.input / histogram.input
        Flow
                histogram.ouput / display_pixel_image.histo
}

Composite Operator { name display_image
        comment "highest level: display any type of data"
Functionality display # needed because it will be called
        Input Data
                GData name input_image
        # No real output except display on screen ...
        Body  display_pixel_image | display_float_data
        Distribution
                display_image.input_image / display_pixel_image.input
                display_image.input_image / display_float_data.input
        Flow
                histogram.ouput/ display_pixel_image.histo
}
```

Definition of Criteria Rules

We first present the common syntax of all types of PEGASE's rules, then we detail the specifics of each type.

Remarks:

- It is often possible to insert a piece of code (in C++) either in premise or in actions of the rules. But it may lead to problems when changing the language or even the machine or compiler.
- Premise conditions and actions of a rule are linked by logical *"and"* (but no "or"), which are represented by commas in the premise or action lists.
- Some functions are specific to one type of rule, but may sometimes be used in the other types.
- Access to arguments of the current operator in one of its rules use the COMPOSEDIDENT notation, but the name of the current operator is implicit. For instance the term `data1.attribute2` in a rule is equivalent to `current_operator.data1.attribute2`, if data1 is a data argument of the current operator. It is also possible to access to local variable defined in the **Let** part (local to the rule, typed, and used for intermediary computation purposes).

```
rule_list : rule
          | rule_list rule
rule : Rule { rulebody }
rulebody : name IDENT comments owner
               Let declslist                    for local variable declarations.
               If precslist Then actslist
         | name IDENT comments owner
               If precslist Then actslist
owner :                                         can be deduced, thus can be omitted
         | owner IDENT                          operator to which rule is attached

declslist : decl
          | declslist , decl
decl : IDENT a IDENT
     | IDENT an IDENT
     | IDENT in IDENT
     | IDENT COMPOSEDIDENT                      COMPOSEDIDENT represents an access to
                                                an object/argument attribute or a method call.
```

Premise

```
precslist : true
          | prec
          | precslist , prec
prec : ( prec )
          | not prec
          | { CODE }
          | rule_exp <> nil                    can be used to test a collection
          | rule_exp == nil                    can be used to test a collection
          | rule_exp compar rule_exp
          | valid IDENT
          | assessment_premise

rule_exp : ( rule_exp )
          | value
          | max ( rule_exp . rule_exp )
          | min ( rule_exp . rule_exp )
          | rule_exp + rule_exp
          | rule_exp - rule_exp
          | rule_exp * rule_exp
          | rule_exp / rule_exp
          | rule_exp quotient rule_exp
          | COMPOSEDIDENT
```

- Usual comparators (<, >, <=, >=, etc.), as seen before, can be used in the *premise* of all the rules, they accept two (numerical) arguments, compare their values and return a boolean. <> and == also accept symbolic arguments or strings.

- **not** indicates the negation of the following condition.
- C++ code is written between '' and ''
- == **nil** and <> **nil** test if an expression is (not) empty.
- **valid** tests if its argument exists.
- Usual numerical operators (-, +, /, *) can also be used in the *premise* of all the rules, they accept two (numerical) arguments.

Specific rule vocabulary for program supervision is indicated in this font. Assessment specific conditions in premise check the judgements that have been passed on either the arguments or the operator by evaluation criteria. They often assess the actual results of their operator after its execution/decomposition.

assessment_premise : assess_operator? IDENT IDENT
| assess_argument? IDENT IDENT
| assess_data? IDENT IDENT
| assess_parameter? IDENT IDENT
| previous_assessment IDENT IDENT
| nb_previous_assess IDENT IDENT compar INTEGER

- assess_operator? tests if the judgement of an operator (first IDENT) is equal to a symbol (second IDENT).
- assess_argument?, assess_data? and assess_parameter? test if an argument/data/parameter (first IDENT) judgement is equal to a symbol (second IDENT).
- previous_assessment checks if a symbolic judgement (second IDENT) has been already attached to the current operator or one of its sons (first IDENT).
- nb_previous_assessment checks how many times a judgement (second IDENT) has been already attached to the current operator or one of its sons (first IDENT) and compares it to a number.

Actions

actslist : act
| actslist , act *The comma is a compulsory separator*

act : common_actions
| choice_actions
| optional_actions
| initialisation_actions
| assessment_actions
| adjustment_actions
| repair_actions

Code parts, assignments, and calls to the (predefined) `display` method are allowed in all kinds of rules.

common_actions : **CODE**
| IDENT **:=** exp
| COMPOSEDIDENT *Method call*
| COMPOSEDIDENT **:=** exp
| display IDENT *IDENT = name of an argument of current op.*

Choice Choice rules conclude on the use (or rejection), among all the available sub-operators, of the ones which are the most (less) pertinent.

choice_actions : use_operator IDENT
| use_operator_of_characteristic IDENT
| refuse_operator IDENT
| refuse_operator_of_characteristic IDENT

- use_operator and refuse_operator allow to prefer or to refuse a given operator. A refused operator is no longer examined for the current execution of the alternative decomposition.
- use_operator_of_characteristic and refuse-operator_of_characteristic allow to prefer or to refuse operator that have specific characteristics.

Example

Choice criteria # of previous composite display_image
Rule { name choice_pixel
comment "If pixel image, display it as is"
If input_image.sort == pixels
Then use_operator display_pixel_image
}
Rule { name choice_float
comment "If float data, discretize"
If input_image.sort == float
Then use_operator display_float_image
}

Optional Optional rule actions decide if a sub-operator must be executed.

optional_actions : use_optional_operator IDENT

Example

```
Composite Operator { name global_search
...
      Body [ visualisation ] - compute_threshold
Optional criteria for visualisation
Rule { name opt_visu
      Let c a Context
      If c.systematic_dispaly == yes
      Then  use_optional_operator visualisation
      }
...
}
```

Initialisation Initialisation rule actions initialise the values of input parameters before an execution or a decomposition.

initialisation_actions : IDENT <- LIST
| IDENT <- STRING LIST
| COMPOSEDIDENT <- LIST
| COMPOSEDIDENT <- STRING LIST
| IDENT <- interval
| IDENT <- STRING interval
| COMPOSEDIDENT <- interval
| COMPOSEDIDENT <- STRING interval
| init_child_parameter COMPOSEDIDENT IDENT

NB: In these rules 1st IDENT is the name of a parameter and COMPOSEDIDENT represents an access to a parameter (sub)field.
When the choice of a value is left to the user, the expert proposes a list or an interval of values and possibly a specific question (STRING).

- <- is used to ask the user for the value of an argument among a list of values or in an interval (*i.e.* initialisation by the user).
- init_child_parameter is used to assign to a parameter of a sub operator (COMPOSEDIDENT stands for suboperator_name.subargument_name) the value of a parameter (named IDENT) of the parent (current) operator.

Example

```
Initialization criteria
Rule { name init
       If true
       Then  f.path := "/u" ,
             y := {{compute()}} # function call
}
Rule { name init2
       If true
       Then  yyy <- "Value of y?" [ 1.0 , 10.5 ]
}

Initialization criteria # of previous operator histogram
Rule { name r_init
       comment "initialize parameter with usual values"
       If true
       Then
             y_min :=  0 ,
             y_max :=  256 ,
             step :=  256
}
```

Assessment Actions of assessment rules detect and diagnose a problem that has to be repaired (locally or somewhere else). They allow to pass symbolic judgements on data, parameters, or an operator as a whole.

assessment_actions : assess_data IDENT IDENT
| assess_parameter IDENT IDENT
| assess_operator IDENT IDENT
| assess_data_by_user IDENT LIST
| assess-parameter_by_user IDENT LIST
| assess_argument_by_user IDENT LIST
| assess_operator_by_user LIST LIST
| assess_data_by_user IDENT STRING LIST
| assess_parameter_by-user IDENT STRING LIST
| assess_argument_by_user IDENT STRING LIST

- assess_data and assess_parameter pass on a data/parameter (first IDENT) a symbolic judgement (second IDENT).

- assess_data_by_user, assess_argument_by_user, assess_argument_by_user are similar but the value of the judgement is given by the user, among a LIST provided by the expert.
- assess_operator gives the current operator a symbolic global judgement (first IDENT) and an indication (second IDENT), equal either to "repair" or to "continue". Only "repair" judgements will trigger repair rules,
- assess_operator_by_user is similar, except that the value of the judgement is given by the user, among a LIST (1st one) provided by the expert the second LIST is composed of "repair" or "continue" symbols that match with the judgements in the first LIST.

Example

```
Assessment criteria # of compute_histogram operator
Rule {
        name r_repartition
        comment "check for regular divisions in histogram "
        If histo.nb_empty_classes / histo.nb_classes > 0.5
        Then
                assess_operator bad_repartition repair # problem detected
                                # and named "bad_repartition"
        }
```

Repair Repair rule actions express how to transmit a problem in the operator hierarchy.

```
repair_actions : re_execute
        | send_up IDENT
        | send_operator IDENT IDENT
        | back_choice IDENT
        | send_down IDENT IDENT
```

- re_execute forces the re-execution of a primitive operator, this triggers its adjustment criteria, in order to assign new values to parameters.
- send_up sends the current symbolic diagnosis (IDENT) to the parent operator of the current one.
- send_operator sends a symbolic diagnosis (second IDENT) to a specific operator (first IDENT), which must *at run-time* have been previously applied in the current execution.
- back_choice returns to the last temporal choice *at run-time* too.

- send_down sends a symbolic diagnosis (second IDENT) to a specific child of the current operator (first IDENT).

Example

```
Repair criteria # of histogram operator
Rule {
      name r_repair
      comment "re-execution of the histogram operator"
      If assess_operator? compute_histogram bad_repartition
      Then
            re_execute # this triggers adjustment criteria
      }
```

Adjustment Action of adjustment rules express a way to repair locally a problem, by re-running the current operator with modified parameter values, after a negative evaluation. The problem may have been detected in another operator.

```
adjustment_actions : adjustment_method
            IDENT                          1st IDENT is the name of a parameter,
            IDENT                          2nd one is the name of its adjustment function.
      | IDENT adjust_by_user interval
      | IDENT adjust_by_user STRING interval
      | IDENT adjust_by_user LIST
      | IDENT adjust_by_user STRING LIST
      | adjustment_step IDENT exp          new step value
      | increase IDENT                     default: increase by one
      | increase IDENT step number
      | decrease IDENT
      | decrease IDENT step number
```

NB: Here the 1st IDENT is always a ***parameter*** *name*

- adjustment_method provides an expert-defined adjustment function. The default function increases or decreases the value by 1 (addition or subtraction for numbers and move one step further or back in the range for symbolic values). Note that **range is compulsory** if you want to use an adjustment method. Percent, addition and dichotomy methods are provided for integers (`percent_integer, addition_integer, dichotomy_integer`) and for floats (`percent_float`...).

- adjust_by_user is used to ask the user for a new value of an argument among a specified list of values or in an interval. A default message is displayed; it may be changed by the expert (via STRING). The arguments are not automatically displayed on the screen, if you want a display, use the `display` function.

- adjustment_step assigns the value of the step of the adjustment function. By default, the step value is 1.
- increase and decrease are used to increase (resp. decrease) the value of a parameter, using the adjustment method associated with the parameter, by default the increment/decrement step is 1, another value may be provided. Note that in this case the **range is compulsory**.

Example

```
Adjustment criteria
Rule {
      name r_min_max
      comment "change min and max ..."
      If assess_operator? histogram bad_repartition # problem!
      and histo.holes() # method holes of type Histogram returns true
      Then
            y_min := item 1 (Float) in "res"
                  # res is the name of the file filled by meth holes with new
                  # min and max; the item is cast to be a Float because y_min is
            y_max := item 1 (Float) in "res"
      }
```

Chapter 6

Methodology of Knowledge Base Development with PEGASE+ and YAKL

We propose a simple methodology, based on our practice of the system, to develop a knowledge base for PEGASE+ (thus suitable for a hierarchical way of expressing knowledge).

The general idea is the following: the expert first describes the best known knowledge, the surest one, *e.g.,* operators corresponding to library programs, a well-known sequence of operators, and so on. This knowledge is often easier and more natural to express for the expert and remains relatively stable. Afterwards the expert can continue with criteria definition (rules), data description, and domain objects. Data and domain objects supply the information needed in rule definition and for knowledge base refinement. Some restructurations are often necessary during this development process.

6.1 First, Define Sub-Problems

When designing a knowledge base to solve a complex problem, it is suggested to divide the whole problem into smaller sub-problems. Each sub-problem correspond to a sub-hierarchy of operators, which may become more tractable than the global one. Moreover it is easier first to describe the frames (operators, data, etc.), and then to decide which rules to attach to which frame. The frame part is often more easy and natural to express for the expert and remains relatively stable.

The connections between the sub-problem hierarchies could be done later, but care must be taken *e.g.,* to keep types consistent among the input/output arguments of different sub-problems.

1. Primitive operators are the simplest and easiest way to define small sub-problem representations, they can be easily defined at the begining of the KB development. Their definition includes input/output data, input parameters (numeric/symbolic information), calling syntax, and some -simple in a first time- initialisation, adjustment and evaluation criteria. This step provides the connection between the knowledge base and the concrete programs.

2. It may appear, while defining the operators, that it is necessary to define some data types (and domain object types) that are relevant to the application. They may be refined later. Most of the decisions concerning the operators and their parameters will initially be taken based on the information contained in these data and domain object types. For instance, in image processing one must explicitly store the information of the contents of an image in such a description, afterwards it is possible to write rules such as: "if the amount of noise is stated in the description of an input image the size of a smoothing filter can be initialised".

3. In some cases however, it is more convenient to describe a high-level problem decomposition and then to focus on each sub-problem and to refine its structure. A high-level sub-problem often corresponds to an interesting functionality for the global problem (from the end-user point of view); its definition starts with the definition of a functionality. This step provides the more abstract view of a sub-problem.

4. In between those extreme positions, the definition of an intermediate sub-problem corresponds to:

   - the definition of a root composite operator and its descendents, this step supplies control information and connections with the real programs, The composite descriptions include input/output data, input parameters (usually more symbolic information at composite level), a body, and some choice/optional criteria (usually defined top-down *i.e.* from composite to primitive operators) and assessment ones (usually defined bottom-up)
   - the refinement of all the associated criteria (and consequently the refinement of some data types if necessary); this step refines the previous information.

In higher level descriptions (composite operators) input parameters may be more or less symbolic information, while in primitive operators at the lowest level, they are more likely to be numeric/string information.

6.2 Second, Add Strategic Criteria

The next step is the creation of the criteria rules which express the approach used by the specialist to execute a plan. The knowledge structures provided by PEGASE enable the systematic expression of the "rules of thumb" and approximate reasoning which are crucial to successful problem-solving.This is done by means of the different kinds of rules discussed earlier. A mixture of numeric and symbolic reasoning maybe required in formulating them.

- Choice rules, usually being the simplest, are defined first. They use the values of object or data fields, data definitions, constraints in the request, etc. to select an operator among the available choices.
- Initialisation rules are defined next. These may be somewhat more difficult, in that they have to formalise the "rough initial guesses" that the specialist makes.
- Adjustment rules are then defined for operators which have adjustable parameters. Step sizes for parameters have to be carefully chosen so that the change in the behaviour of the algorithm is neither too sudden nor too gradual.
- Then evaluation rules are defined. This is rather difficult a task, since the question "Are the results good enough?" is often subjective, and appropriate quality measures may not be readily available. However, a close examination of the specialist's problem-solving technique often reveals hidden reasoning capable of being expressed in concrete terms as evaluation rules.
- Finally, the repair rules are defined. These rules determine the overall failure-handling mechanism, and are crucial to the success of the application.

6.3 Complete and Refine the Base

Based on the sub-problem hierarchies constructed in paragraph 6.1 it is now possible to fully determine the entire operator hierarchy. The best way to do this is to look at the entire hierarchy as one big problem with several predefined sub-problems (the ones just constructed).

1. Starting from low-level sub-problems one can then define more complex ones by gluing sub-problems to each other. This gluing will be done by defining the data flows between them and by establishing a control among them. The same sub-problem may appear in several higher-level problems. This is the bottom up way. The advantage of a pure bottom-up approach is that at each stage, one actually has constructed an operational (though partial) knowledge base, that can be tested in order to correct small mistakes.

   The decisions concerning the operators and their parameters can be based on results of previous operators. The data flows are the means to incorporate such contents dependency in the KB.

2. The final refinements consist of

   - Refinement of criteria rules, to choose among alternative, to initialise parameters, etc. Rules often rely on data types or domain objects ones, so the detailed descriptions of types can be done at the same time you construct the rules, otherwise you can describe more information about some data than necessary. The easiest rules are initialisation ones, then adjustment on parameter values. Repair and evaluation are more complex to define. When initialising or adjusting a parameter it is often possible first to reason with symbolic information, and then to translate the symbolic information into real numbers.
   - Refinement of information very dependant on the application domain, such as functionalities, characteristics - preconditions, postconditions, I-O relations, effects, w.r.t. data types and domain objects. Pre and post-conditions define the semantics of an operator.
     - Pre-conditions on input data description allow the execution of an operator.
     - Post-conditions may be used to check the contents of output data descriptions. So you have to check at the end of the development if all the postconditions that should be mentioned are really mentioned. Otherwise you can end up with a partial description.
     - I-O relations store information on the results of an operator (*e.g.,* change of size of an output image w.r.t the input image,... Information pieces that are the same as in the description of the input arguments *must* be mentioned *explicitly* in I-O relations, otherwise they are lost for the forthcoming programs.
   - Refinement of information about links among operators (usually done at the end, when one is more sure about the KB organisation): data distribution and data flows.

   This phase is a constant back-and-forth cycle within the different abstraction levels, and also in the data descriptions.

Of course, the way of expressing knowledge is not always optimum and some things remain rigid, due to remaining problems or decisions made.

6.4 Guidelines

Here are some "rule of thumb" that can guide the KB construction:

- A parameter is something that tunes operators behaviour but not its functionality.
- Try to keep operators unaware of their possible combination in a sequence, choice, etc. That is, names should be chosen w.r.t. the method an operator represents. Operators

should be self-contained. Father operators have knowledge about their decomposition, but son operators should not have to know about their parents. So keep information concerning a sequence (i.e. a history) of operations, at the level were the links between operations are expressed (ie. in composite operators).

- Keep low-level information at low-level (*e.g.*, numeric value of parameters of primitive operators). If you want composite operators to be reusable, you have better not to express information about sons in a too precise way. An idea is to use symbolic information about the way the decomposition must be done at composite operator level, and that this information is "translated" in a numeric/more precise way inside son operators. This is useful for parameter initialisation/adjustment, assessment and repair, choice, etc. *E.g.*, use predicates as `use_operator_of_characteristic`, or symbolic values as "small", "big", etc.

- In many cases the reasoning for initialisation is performed in two steps. The first step is reasoning with symbolic information, and the second step is a translation of the symbolic information into real values. Knowledge for the first step must directly be elicited from the expert, while the knowledge for the second step can come from the expert (experience, heuristic values), can be calculated, or can be based on tests.

- Keep in mind this basic principle: a father operator knows about the organisation of its sons and about the behaviour it expects from them. In spite of this "paternalistic" view, son operators should be able to work on their own, thanks to the knowledge about their own behaviour. The abstract description of a program should allow it to be applied in different situations.

- It is very difficult to define a knowledge base independently from an application domain (*i.e.* so that the KB can be completely reusable). If, for example, a knowledge base is constructed for a specific application in image processing (*e.g.*, road scenes), this specificity prevents it from been reused "as is" in other applications (*e.g.*, underwater scenes), because specific information could be stored in the description of an image. Keeping that point in mind when writing a KB, allows experts to disconnect as much as possible specific features from reusable ones, and makes the shift from one application to another easier. For example the definition of a sub-type of `Image` with specific attributes (*e.g.,* application_domain) and tests limited to this attribute (*e.g.,* **If** image.application_domain == road scenes **Then** detect-cars) may restrict the scope of the changes.

- About rules

  - The best is to try to define the rules in a bottom-up way, that is to begin with the rules related to the simplest sub-problems, and in particular primitive operators, because they are closer to the real programs. The information is easier to get and safer (less prone to interpretations and variations). Before defining rules at

higher level, you must rely on the available low-level rules, otherwise some high-level rules may never be implemented, for lack of existing means of obtaining the necessary information.

- In order to obtain general and clear rules, prefer fine-grain ones. Fine grain rules cannot be decomposed into smaller ones. Coarse grain rules are reasoning short-cut that don't take into account the exceptions. Intermediate symbolic values may be useful in defining fine grain rules.
- Defining rules may lead to defining some additional operators (*e.g.*, to compute an initial value), which modifies the hierarchy. The rules may also lead to modifications of data and domain object descriptions, because while constructing the rules one can see on what information the decisions are based. The modifications mean to enrich them or, on the contrary, to remove useless attributes, not relevant for program supervision reasoning.

6.5 Validation Facilities

Once the hierarchy seems terminated, you still have to test it against several requests on different input data. The resulting KBS must be able to adapt itself to produce different plans, in different situations. If it is not the case, errors should be detected, using the "trace" facilities.

A good idea is to prepare test-beds, from the beginning of the KB development, which are significant for the application. The test-beds may be kept all along the life of the KBS, as reference tests, because a KBS is prone to change!

6.6 Requirements for Using Program Supervision Techniques

This section analyses which properties the programs and their arguments must verify in order to be candidates for re-use with program supervision techniques.

6.6.1 Program properties

If there is only one unique and clearly defined functionality for each program the model is directly applicable. A primitive operator is thus created for each program. If it is not the case, that is if one program achieves several distinct functionalities, the solution is either to rewrite the program in order to split it into smallest ones, one per functionality, or to define as many knowledge base operators as there exist sub-functionalities in the program.

Moreover, if the programs are already managed by an "interpretor" such as a command language or a graphical interface, additional work is necessary to solve the communication problems between a program supervision system and the individual programs.

6.6.2 Argument properties

Programs can only be re-used if they do not work with "magic numbers" *i.e.* fixed values for important internal parameters that have been obtained by past experiments. So, explicit internal parameters for each program by rewriting and adding explicit arguments. The same problem may arise with data which may be implicit, *e.g.,* in the case of programs communicating via a shared memory. There are two possible solutions: the first one is to rewrite the program and to create new arguments for all data. A second solution is to keep the use of a shared memory, for efficiency reasons, but to represent explicitly the implicit arguments in the primitive operator corresponding to the program.

6.6.3 Composite operator properties

Introducing a first abstraction level is natural when there exist several alternative primitive operators sharing the same functionality. The solution is to create in the knowledge base one composite operator per functionality. The decomposition of this composite operator of choice type and the sub-operators are the alternative primitive operators.

When typical program combinations are available (*e.g.*, shell scripts, with sequences, alternatives, etc.) this information can be directly described and represented in the knowledge base by creating one composite operator per typical combinations.

The knowledge base can contain several abstraction levels when the body of a composite operator is itself composed of other composite operators.

6.6.4 Criteria properties

The criteria are not mandatory and each operator must not contain all types of criteria. The repair knowledge for instance can be located only in a few precise operators. Even if the knowledge representation of the operators is homogeneous, their usage is very dependent on the knowledge to express.

The criteria can manage the degree of interactivity with the user. If there exist methods for automating the computing of values (parameter initialisation methods, parameters adjustment methods, or methods for the evaluation of the results), these methods can be directly translated into specialised criteria. If these methods do not exist, specialised criteria can nevertheless be created to guide the interaction with the end-user. It is especially useful for results evaluation: the role of the criteria can be limited to the automatic display of some output data and of a list of possible assessments which are compatible with the repair knowledge. The user only selects a particular assessment for the displayed results.

6.7 Summary

It appears that depending on the set of programs to supervise the knowledge modeling effort is more or less important. These remarks lead to a coarse methodology of knowledge base building: the easiest way is to begin by describing concrete individual programs, then

to create higher levels of abstraction using composite operators. Criteria may be added afterwards, the more criteria the knowledge base contains the more efficient and flexible the program supervision process will be.

Chapter 7

Example of Knowledge Base Development

This chapter details the different steps indefining a knowledge base using YAKL and following the methodology previously described. It also states the main points to rememeber for experts who want to use YAKL.

The expert, who is the knowledge base designer, defines YAKL files (with the `.yakl` extension). Usually one file corresponds to a logical unit, such as a functionality and the operators that achieve it. The emacs mode (see section 10.3.1) can help to respect the syntax of the language. Each file must declare as "imported" the files containing entities it refers to (*e.g.*, types of data, primitive or composite operators, etc). Files must be parsed in semantical order *i.e.* files defining new argument types before files containing operator descriptions with arguments of these types, files defining primitive operators before files containing composite operators that use these primitive ones in their bodies, etc. The `Makefile` files provided in the distribution can help in doing this.

YAKL may be used in its simplest form to document program sources (by means of primitive operators) and also a few usual ways of combining the programs (by means of composite operators).

Here follow some steps to go from the simplest to the more complex use of YAKL in order to either document or even design an operational knowledge base, *i.e.* that may be used by an engine like PEGASE+ to actually perform program supervision.

7.1 Starting a new Knowledge Base

Following section 6.1, a new knowledge base starts with the definition of sub-problems, that is the *operators* which represent the most important part of a knowledge base in program

supervision. It is usually easy to begin with describing primitive operators and even some composite ones.

7.1.1 Defining a Primitive Supervision Operator

A primitive operator usually corresponds to one existing program in the library, achieving a processing function (or functionality in program supervision vocabulary, *e.g.,* segmentation in image processing). The operator describes the program as a "black box" only known by some information on how it can be used in different situations and by its inputs and outputs. In some cases, when programs are very big, it could happen that one program achieves several distinct functionalities, the solution is either to rewrite the program in order to split it into smallest ones, one per functionality, or to define as many knowledge base operators as there exist sub-functionalities in the program. We can note that, as a side-effect, the building of a program supervision knowledge base may have an influence on the methodology of code design, resulting in more modular and structured codes.

Each supervision operator in the knowledge base has a name, in order to identify it, any identifier is accepted. In the example below, the primitive operator corresponds to a program that performs an oblique rotation on images, so its name has been chosen accordingly. A primitive operator also indicates an author, who is the author of the corresponding program.

In the following, YAKL keywords (remind that they are reserved words in the syntax) are in bold face (and the concatenation of all YAKL lines constitutes an acceptable definition).

Primitive Operator {
    name obliqueRotation
    authors "Mr. B."

An operator also has symbolic characteristics, that may help the selection process of the engine, and input and output arguments, with an associated type. In the following example the operator uses an "oblique analysis" method and detects from 1 up to 10 factors, that is why its characteristics are "oblique, 1_10_factors". These symbols can be used in choice criteria for example (by calling `[ref]use_operator_of_characteristic`, see pages 93 and 73).

Characteristics oblique, 1_10_factors
Input Data
    File **name** reducedSequence
Input Parameters
    Integer **name** nbFactors
        comment "number of factors to look for"
        default 3
        range [1 , 10]
Output Data
    File **name** factorCurves
    I-O relations
        factorCurves.path := reducedSequence.path ,
        .....

In this first example, we have only used predefined types for arguments. Predefined types are `Integer, Float, String, File` and `Image`. `Image` is currently defined as a sub-class of `File`, so both have the following attributes (of type `String`): `basename, path` and `extension`. They also have predefined methods: `get_filename()`, which returns the complete name of the file (including its path) `get_basename()`, which returns the base name of the file, and `get_extended_name()`, which returns the base name plus the extension of the file. The `Image` type has two additional slots: `x_size` and `y_size` of type `Integer`.

YAKL distinguishes two categories of arguments: data and parameters. Data arguments have *fixed* values which are set or computed (via the data flow), while parameters can be tuned. An operator *cannot* modify its input data arguments, but it can modify its input parameters using initialisation or adjustment criteria (see pages 100 and 109). A data argument refers to a global instance of a data object in the fact base (see page 94).

Each argument (data or parameter) has a name. The names of the arguments belonging to the same operator *must* be different. An argument may have an optional attached comment. The operator of the example has one input data argument, one integer parameter and one output data.

Parameters usually belong to a simple type (integer, float or string), because for other (structured) types it is not clear to decide what does "to adjust the value" mean. Parameters may be described in more details by a default value (3 for `nb_factors` in the example), that will be used when the engine needs to use the parameter value and when no exact value is available. The default value constitutes a simple way to initialise a parameter, usually this value is an average one, that is supposed to work in most of the cases. They may also have an optional range of possible values. It may be an interval for numerical parameters (as in the example) or an enumeration of values for all simple types (numbers and strings). They may also have

To each output data *of primitive operators only* the expert may associate optional (but recommended) so-called "I-O relations", which describe relations connecting it with the inputs. This applies only to structured type (for instance `File` or `Image` in the predefined types), but not to simple types such as integers, floats or strings. This part may express logical relations, thus giving some hints about the corresponding program behaviour (*e.g.,* "the size of the output is half of the size of the input") or more pragmatic ones, as in the example, which writes that the path of the output file is the same as the one of the input file. Output data arguments are computed during the reasoning process mostly by means of these I-O relations when executing primitive operators. The expert may also use I-O relations to impose some values on the output data (*e.g.,* location of a file).

In order to actually call the real program, the engine first checks if the operator preconditions are achieved. Preconditions (like postconditions) are a list of boolean expressions, separated by commas (,). In the example, we only mention a very simple condition, checking if the input file exists on the disk, represented in the language by the keyword `valid`.

Preconditions

valid reducedSequence

The descriptive knowledge contained in this common part of operator descriptions is used by the program supervision engine during planning. For example, information on the functionality, characteristics, description of arguments, and pre/post-conditions of operators is useful for selection purposes, while I-O relations maintain the continuity of data contents along the reasoning, after each execution.

Provided that its execution preconditions are true, the execution of a primitive operator corresponds to the execution of its associated program. A primitive operator thus describes the necessary calling information. The only mandatory information is the language (*i.e.* will the program be called via the shell or is it an internal C++/Lisp program) and the calling syntax (or command line). Commands in this line are either procedures written in the language of the engine (here C++/Lisp) or shell scripts to be run under the operating system of the machine. In the first case the procedures are usually written in so-called code files, attached (and linked) to the knowledge base (see page 103), in the second case the shell scripts must be executable. The additional indication of "type" (real or simulation) is for the engine to know which type of execution is wanted: "real" is the default, and means a real execution of the program, while "simulation" means that the syntax will be printed but not executed. The latter case may produce scripts which could be run afterwards on different sets of data.

Call

language shell
syntax cobl reducedSequence.get_filename() nbFactors **endsyntax**
program name cobl
type simulation

}

The syntax is composed of fixed parts, that must be kept as is (like `cobl` in the example) and of parts that will be instantiated by the actual values of the input/output arguments for execution (like `nbFactors` in the example). The parts of a calling syntax which are equal to (or contain) the name of one of the current operator arguments will be interpreted and translated as an access to this argument. So don't use an argument name that may be confused with a calling argument for instance, which must be translated as is. For instance, if the action to execute is a shell call to `tar` don't use an operator argument named `c`, because it is a calling argument of tar, and **syntax** `tar c` **endsyntax** must be translated into `tar c` and not into `tar current_operator.c`. In particular, composed identifiers (like `reducedSequence.get_filename()` in the example) are by default understood as access to attributes or methods of the corresponding operator argument. If you want to avoid this translation, just quote the tricky parts of the command line (by enclosing them into two quotation marks).

It should also be noted that all the parts of the calling syntax must be of *numeric, symbolic or string* type, so that they could be either printed or passed to the shell.
Here `reducedSequence.get_filename()` stands for the complete name of the file (which is a string) (`get_filename()` is a predefined method on the `File` type). `reducedSequence` alone would have generated a compile *error*, because the program supervision engine does not know how to print a file object.

Remember:

- choose a (significant) name,
- provide a sound comment for documentation,
- decide which input arguments should be data (non adjustable) or parameter (this is usually not straightforward!),
- choose types for input and output arguments (it is always possible to refine a type, at first choose predefined ones or define new simple ones: see page 94)
- collect calling information (options, arguments, etc.) and decide the calling syntax accordingly (in a printable form, with no confusing names of arguments versus calling fixed parts).
- define simple I-O relations and conditions

See pages 64 and 66 for details.

7.1.2 Defining a Composite Supervision Operator

Once the primitive operators are described, one may want to manipulate "packages" of programs. Indeed, it may be convenient to gather in one entity some usual sequence of programs or all the programs that perform more or less the same function with slight differences, etc. This is done via composite operators.

The description of a composite operator starts like the description of a primitive one: *i.e.* a name, inputs, outputs, preconditions, etc.

Composite Operator {
    name poly
    comment "solve polynomial systems"
Input Data
    File **name** Sy
Output Data
    File **name** sol
Preconditions
    valid Sy

The way a composite operator has to be refined is expressed in its body which contains a description of its decomposition into sub-elements, *i.e.* names of its sub-elements and the control over them (which is the type of decomposition), and data flow information between parent and children (named **distribution**) and between children in a sequence (named **flow**).

The allowed types of decompositions are the alternative (indicated by |), the sequence (indicated by **-**) and the parallel (indicated by ||). The most usual decomposition types are

sequences and alternatives. A sequence decomposition indicates that all children links have to be "executed" in order to consider the parent as executed. On the contrary, alternative decompositions represent choice between different ways to solve a problem. An alternative decomposition parent is considered as executed as soon as one of its children is. These decompositions at different levels of abstraction must end on primitive operators. The sub-operators (or sub-functionalities in the future) *must* have been previously defined (they must be known by the parser). The same primitive or composite operator may appear in different decompositions.

Sequence Decomposition

In the following example, the body describes a sequential decomposition among two sub-operators: `dimension` and `solver` (but of course a sequence may contain more than two sub-operators).

Body

dimensions - solver

Each operator (and the same is true for functionalities) has its own naming for its input/output arguments which are considered as formal variables. In a composite operator all arguments –in parent and children– referring to the same object or value must be connected together, *i.e.* formal variables must be unified. That is why describing a decomposition of sub operators implies to describe also how the data are transmitted between a parent and its children, as well as between children in a sequence.

The connections between the formal arguments of a parent node and those of its children are denoted by the term **Distribution**, while the connections between brothers in a conjunctive link are denoted simply by **Flow**. After connection, a change to the common actual value is passed on all the arguments connected to it. The syntax of the **Distribution** and **Flow** sections is similar to the one used in logic for substitution of formal variables (and in fact we address the same issue). It has the following form:
$OperatorName\ .\ ArgumentName/OperatorName\ .\ ArgumentName$ (note the blanks around the dots).
In the **Distribution** section, the first $OperatorName$ must be the name of the parent operator. Both types of flows connect data arguments respecting their roles (*i.e.* they are consistent with respect to the inputs and outputs). That means that in the **Distribution** section, inputs of the parent operator have to be connected with inputs of children (and the same for outputs), while in the **Flow** section, a data flow only connects output arguments of an elder child to input arguments of a younger one.

Distribution

poly.Sy / dimemsions.PSin
poly.sol / solver.sol

Flow

dimensions.PSout / solver.Sy

}

Remember:

- Sub-operators must already have been defined.
- All arguments must be connected by at least one data flow link (with the correct in/out modes and type compatibility).

See page 68 for syntax details.

Alternative Decomposition

Here is another example of a composite operator with a body describing an alternative decomposition among two sub-operators: `principalCompAnalysis` and `correspFactorAnalysis` (but of course an alternative may contain more than two sub-operators).

Composite Operator {
name orthogonalDecomp
Input Data
File **name** analysisArea
Input Parameters
Symbol **name** metric
comment "metric to compute distances"
range [ki2 , identity **]**
Output Data
File **name** filteredArea

Preconditions
valid analysisArea
Postconditions
valid filteredArea
Body
principalCompAnalysis | correspFactorAnalysis

The data flows are simpler than for sequences because there is no flow among children. They only consist of the **Distribution** part in alternative decompositions, like below:

Distribution
orthogonalDecomp.analysisArea / principalCompAnalysis.analysisArea
orthogonalDecomp.analysisArea / correspFactorAnalysis.analysisArea
orthogonalDecomp.filteredArea / principalCompAnalysis.filteredArea
orthogonalDecomp.filteredArea / correspFactorAnalysis.filteredArea
}

Summary about Flows

Figure 7.1 summaries the data flows for both sequences and alternatives. Parallel decompositions follow the same scheme as alternatives.

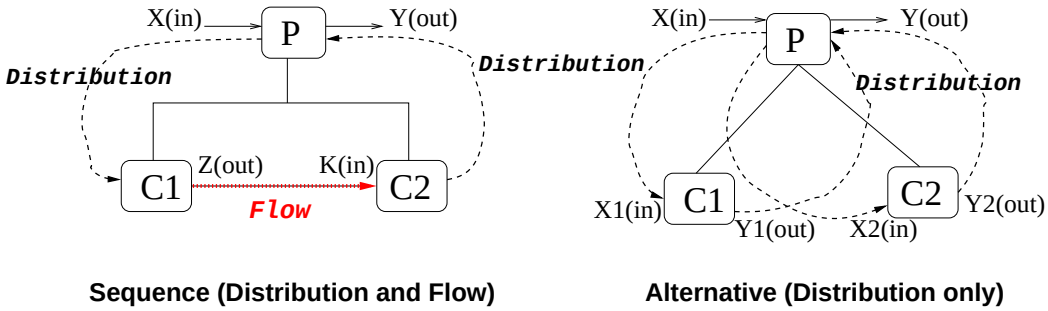


Figure 7.1: Data flows (between parent and children or between children)

Remember:

- Sub-operators must already have been defined.
- Don't forget choice rules for *all* alternative sub-operators.
- Data flow connections (distribution) must exist with every sub-operator (*i.e.* the input data of the parent must be connected with the input data of each child, and the same for output).

See pages 68 and 73 for syntax details.

7.1.3 Defining Types

Predefined types for arguments may soon appear limites, so, in parallel with operator descriptions, structured types can be described for both *arguments of operators* and for *domain objects* that depend on the application. YAKL provides a standard frame-based representation for types with a hierarchical organisation. Both data and domain objects types descriptions can include methods associated with them, *e.g.*, `display` methods.

Argument types are used to type the arguments of operators (and of functionalities), when predefined types are not sufficient.

Extending an existing Argument Type

In addition to the built-in types provided by the system (`Integer, Float, String`, etc., described page 88) operator arguments often require more sophisticated types. The expert has to provide a representation for the new data types that are manipulated by operators. In these descriptions the expert may include some "semantic" information about the important features of the data that are affected by programs (*e.g.*, in the image processing domain theses features may be the size, the noise, the content type, the status, the main object of interest, etc.).

For instance, the `File` or `Image` types, though useful, are rapidly limited for specific applications. The expert can extend these types to fit the needs of the application, like in the two examples below: *NB: we only use the first version ("computer-oriented") of the definition of attributes (see 55) in all the following examples, but the second one would be accepted as well.*

```
Argument Type {
        name MyImage
        Subtype Of Image
        Attributes
        Integer name number
                default 1
        Override String name extension
                calculation calc_fct
        Symbol name physical_process
                default mri
                range [ mri Xray nuclear ]
        Symbol name gradient
                default : unknown
                range : [ low high unknown ]
        Symbol name format }
```

```
Argument Type {
        name PFile
        Subtype Of File
        Attributes
        Integer name nb_variables
                default 1
        Integer name nb_equations
                default 2 }
```

The **Subtype Of** key words indicate the name of the super type. The attributes of the extended type (here `File`) are inherited by the derived type, and the expert may add new attributes (like `number`) or override inherited ones (like `extension`). Overriding only allows experts to modify the **calculation** (if_needed daemon), like the call to `cal_fct` function in the example[1].

After this definition, it is possible for instance, to use the new type for an operator argument, and to use the new attributes in rules, conditions, I-O relations, etc. For instance, in the example of page 91 the change of type of the input data `Sy` to `Pfile` allows the expert to add new preconditions with a *semantical meaning* for the application, and the description becomes:

```
Composite Operator {
        name poly
        comment "solve polynomial systems with at least same
                number of equations than number of variables"
Input Data
        PFile name Sy
Output Data
        PFile name sol
Preconditions
        valid Sy,
        Sy.nb_equations >= Sy.nb_variables
```

[1] In the future it will also be possible to modify default values or ranges of attributes.

It is also possible to define new argument instances (to be used in a request, see page 102), filling the attribute values (both inherited and new ones):

Argument Instance {
MyImage **name** image1
Attributes
number := 2
path := "/usr/home/Images"
format := f2 .....
}

Of course it is also possible to define brand-new argument types, the syntax is the same, but without the **Subtype Of**.

Remember:

- Experts data types may enrich the predefined types.
- Data types describe some semantical information about the program supervision problem of an application.

See page 55 for details.

Defining a new Domain Type

Depending on the application, it may be necessary to define so-called "Domain Types", which correspond to types of objects related to the application domain, that can help in the program supervision process. The aim is **not** to exhaustively represent all the objects in the domain, but only those which have an influence on the engine reasoning (*i.e.* which are useful in some criteria to drive the selections or choices). These objects of interest for program supervision purpose in the application domain may be represented in the fact base. The new types can then be used like other ones to define global instances of domain objects. The instances contain information that can be accessed and modified during reasoning, for example rules may test the values of the attributes of such objects in their premisses or assign them values in their conclusions.

The syntax is almost the same as for argument types, but it should be noted that the domain types are *not* accepted for operator arguments.

Object Type {
name Context
comment: "the context of processing"
Attributes
Symbol **name** user_requirement
comment "user's requirements for processing"
range [fast average any **]**
}

Sub-typing is also possible:

Object Type {
name IP_Context
Subtype Of Context
comment "the context for image processing"
Attributes
Image **name** reference_image
}

After this definition, it is possible to define new domain object instances (to be accessed in rules), filling the attribute values (both inherited and new ones):

Object Instance {
Context **name** userc1
Attributes
user_requirement:= fast
}

Attributes that are of structured types, must be either filled by an instance, which must have been defined beforehand or inline (see 60), here we chose the first option:

Argument Instance {
Image **name** rim1
Attributes
path := "/usr/X/"
basename := "refer"
.....
y_size := 256
}

Object Instance {
IP_Context **name** ipc2
Attributes
user_requirement:= fast
reference_image := rim1
}

If not all attribute values are filled, the YAKL parser will produce a warning, but it may be acceptable for the application: that means that the attributes will be filled during reasoning (otherwise that means that they are useless for program supervision and may be discarded!).

Remember:

- Domain types must contain only information useful for program supervision.
- Sub-typing is possible.

See pages 55 and 60 for details.

It should be noted that all domain objects have a predefined String attribute: `name` which can be referred to in rules, effects, etc., as in the rule:

Rule { name retrieve_by_name
Let c **a** Context
If c.name == ipc2
Then c.user_requirement := fast **}**

7.1.4 Add Strategic Criteria

So far the descriptions of knowledge by means of operators, arguments, etc. provides a rather static view. When this structural part of the knowledge base is well developed, comes the time to think about more dynamic knowledge, that is *criteria*. Criteria provide decision information that will be used by the engine during reasoning.

The operators use various criteria in order to manage their input parameter values (initialisation and adjustment criteria), to assess the correctness of their results (assessment criteria on output data), and to react in case of bad results (repair and adjustment criteria). They play different roles in the reasoning of a program supervision system.

For the time being, the criteria are represented in YAKL by specialised rule bases (groups of rules) which are attached to operators. Initialisation, evaluation/assessment, repair and adjustment rule bases can be attached to all operators, while choice and optional criteria are specific to composite operators.

For all types of rules, premise (**If** part) of the rules are a list of boolean expressions separated by commas (,). In the same way, actions (**Then** part) are a list of expression also separated by commas.

The locality of the criteria allows each piece of the knowledge base to have its own decision knowledge with respect to its role in the knowledge base and the kind of information it has access to.

The easiest criteria to define are choice and initialisation ones.

7.1.5 Choice Criteria

In an alternative composite operator, like `orthogonalDecomp` in section 7.1.2, the program supervision engine must have the knowledge to choose among alternate sub-operators. This is given by experts in choice criteria, in the form of a set of rules (named a rule base) that conclude on the choice (or rejection), among all the available sub-operators, of the ones which are the most (resp. less) pertinent, according to the data description, the context and the characteristics of the operators. This kind of criteria is used for planning purposes. A choice rule for `orthogonalDecomp` is presented below:

```
Choice criteria
Rule {
        name Choice1
        If metric == identity
        Then use_operator principalCompAnalysis
}
```

The rule in the example chooses the sub-operator named `principalCompAnalysis`, if the metric to apply is the identity. Note that, since `metric` is a parameter (see page 93) with no default value, it must have been assigned a value by some initialisation rules.

Choice criteria can also reject operators, or guide the choice towards an operator having some characteristics (matching the **Characteristics** field of operators). For example in image processing, suppose you have two operators that perform stereovision using a pyramidal technique and that are based on contour chain points. The first operator directly implements this algorithm, while the second one first separates the two interlaced acquisition frames by sampling the lines. The second operator is thus well-adapted when motion is present. The rule presented below implements such a choice criterion:

```
Choice criteria
Rule { name  choice_charact
        If motion == present
        Then use_operator_of_characteristic sampling_for_motion
        comment "only 1 of the 2 interlaced frames if motion"
}
```

Here is another example of choice rules, that select or refuse an operator, based in particular on information provided by the end-user about an input data: `in_image.gradient`; `<-` means that the value will be dynamically asked to the end-user at execution time, see page 74:

```
Choice criteria
Rule { name  init_gradient
        If  in_image.gradient == nil
        Then display in_image ,                    Call to the display method
             in_image.gradient <-                  The user is asked to answer
             "Importance of gradient in image background?"
             [low high unknown]                    Possible choices for the user
}
Rule { name  choice1
        If  in_image.gradient <> high
        Then use_operator op1
}
Rule { name  choice2
        If  in_image.gradient == high
        Then refuse_operator op1,
             use_operator op3
}
```

In this example, the premisses cover all the possibilities of values for `in_image.gradient`. Such a completeness is suitable, though not yet enforced by the parser.

7.1.6 Initialisation Criteria

These criteria are composed of rules that will be used by the engine to initialise parameter values before the first execution/decomposition of an operator. They store the knowledge of an operator on how to initialise its own parameters. That is why they are *attached to* operators. They provide the expert a more sophisticated and flexible way to perform initialisation of parameters than the default value mechanisms (which are often "magic numbers" given by experienced experts/users of programs). However, they are not compulsory if all parameters have got a default value. Initialisation rules can perform some computation and several rules may chain together in order to decide which value to assign depending on data, user's requirements, etc.

Parameters may be numeric as well as symbolic ones like `metric` in the example of the `orthogonalDecomp` operator, page 93. Assuming that the input data `analysisArea` is now of type `MyImage`, we can write the following rule to initialise this parameter:

Initialisation criteria
Rule { name init_metric1
If analysisArea.physical_process == mri
Then metric **:=** identity **}**
.....

Here is another example, for the `segm1` operator (see page 102) using a domain object (of type `Context` (see page 96):

Rule { name init_precision
comment "a spot in the image implies a precise computation"
Let c a Context
If c.user_requirement == average ,
im_in.spot_presence == awkward
Then precision := precise**}**

And finally, here follows an example of rule chaining, first reasoning with symbolic information, and then translating the symbolic information to real numbers. An operator which performs a thresholding by hysteresis needs the initialisation of two input parameters: a high threshold and a low threshold. These thresholds are defined as numerical parameters. The initialisation of these values is performed in two reasoning phases. In a first phase only symbolical information is used. For example if the user only wants *few* details to be extracted, the threshold should be *high,* as shown in the rule:

comment "few details implies high threshold"
If context.user_constraints.details == few
Then threshold **:=** high

In a second reasoning phase the symbolic values are translated to concrete numbers, as in the rule:

comment	"average threshold implies 25 as heuristic value"
If	threshold == average
Then	heuristic_value := 25

This second phase of translation into numbers can be simply done by selecting a static heuristic value (as shown in the previous rule) but also by dynamically computing a value using a calculation method (*e.g.*, probability computations based on the image histogram).

7.2 Complete the Base

In order to get a complete KB that can be executed by PEGASE+, some complementary steps are necessary.

7.2.1 Import clauses

When the knowledge base begins to grow it is not possible (and not good programming) to keep all information in one single file. A real knowledge base is usually splitted into several `*.yakl` files. In this case, if for the description of some operator or type the expert needs to refer to previously defined types or operators, he/she must first *import* their files (in a similar way to the #include mechanism in C, except that you may write several names of imported files on one line, *but not on several lines*: one new **Import** key word is necessary at the beginning of each new line). The file names in the list of imported files should be written without the `.yakl` extension.

Example :
Import deftypes primitives1
Import primitives

These should be the first lines of a `.yakl` file, they mean that this file uses some elements that are described in three other files: `deftypes.yakl`, `primitives1.yakl` and `primitives2.yakl`.

7.2.2 Defining a Functionality and a Request

Other important concepts in program supervision are functionalitie. Even if they are by definition, less numerous in a knowledge base than operators, they are compulsory to get an operational base.

A *functionality* represent an abstraction (without details peculiar to an individual implementation) of a processing function. Several operators can concretely realize one abstract functionality. The alternative decomposition type often corresponds to a specialisation in this way, and provides a way of grouping operators into semantical groups corresponding to the common functionality they achieve. This is a natural way of expression for experts and it allows levels of abstraction above specific operators.

The end-user has also to provide a *request* with specific input data, about a particular problem. A request specifies the functionality to achieve along with some initial concrete

input data. That constitute the user's problem statement. At least one request is necessary to trigger the reasoning. The program supervision engine can then more or less automatically process the user's request, using the knowledge base contents, without burdening the user with technical processing problems.

For example, supposing that we have the `image1` instance of `MyImage` defined page 96, we can define a new functionality and a request:

Functionality {
    name Segmentation
Achieved by segm1
Input Data
    MyImage **name** im_in
        comment "initial global image"
Output Data
    MyImage **name** bin_im
        comment "binary extracted image"
}
Request {
    Segmentation **name** segmentation_image1
Attributes
    im_in **:=** im1
}

An operator, like `segm1` below, can then declare to achieve this functionality. It **must** then have the same input and output arguments as its functionality (*i.e.* the same argument names and same argument types, but it may also have additional ones, for instance some parameters). *NB: for the moment the names of the common arguments must be exactly the same, such as* ***im_in****.* A connection with a functionality is only necessary for those operators that will be able to answer a user's request. It is used by the program supervision engine which tries to match the request and the operators in the knowledge base (see chapter 4).

Primitive Operator {
    name segm1
`Functionality :  segmentation`
Input Data
    MyImage **name** im_in
        comment "original image"
Input Parameters
        Integer **name** precision
        ....
}

Remember:

- Define only functionalities that are relevant from the end-user's point of view (those that may lead to users' requests)
- Operators with an attached functionality are entry points for the PS system.
- These operators must conform to the types/names of arguments of their functionalities.

See page 62 for details.

7.2.3 Defining a Knowledge Base

The expert must also provide in *a separate file* with a `.kb` extension (example: `my_base.kb`) a representation of the contents of a knowledge base in program supervision. It refers to a list of files which contain either the description of the knowledge base components (which are `.yakl` files containing the descriptions of operators, rules, etc.), or code in C++ for expert-defined methods and functions necessary in the application domain. These files are assumed to be stored in the same directory (indicated by (optional) **KB Path :**). At a given time, there is only one knowledge base (and one fact base containing domain object and data instances) active in the system.

Kb { name IP
Complete Name "KB for satellite IP"
Authors "X Y Z"
KB Path "/usr/local/bases/IP"
Version 2.0
Root Node IPprocessing
List of Files ip1 ip2
}

The root node, in hierarchical program supervision, as in PEGASE+, is the node which is at the top of the (preferably unique) hierarchy of operators, but you may have several roots in the general case, since there may be several partial hierarchies of operators in the knowledge base.

The optional *KB Path*, if filled, must correspond to an absolute path.

The list of files correspond to all the `.yakl` file that have been defined (the `.yakl` extension is assumed, *e.g.,* here `ip1` stands for `ip1.yakl`.

Note that there is no need to mention the file containing the KB definition itself (i.e. the `.kb` file where the previous lines are written in) in the list of files, provided that this file contains *only* that definition.

Remember:

- A KB is described in a separate file with a `.kb` extension.
- Be careful to write down the right path.
- Update the file names (in case of changes of name, addition or deletion of a file)
- Don't indicate the `.yakl` extensions
- Choose a root node (even if it is artificial, PEGASE+ needs one).

See page 53 for details.

The expert may also write some code files, for instance `.cc` files in C++, that contain the code of methods of types or of additional functions used in rules, in calculation parts, etc.. These files will be linked with the base. They are not necessary at the beginning of a base. The syntax is:

Kb { name IP
..... **List of Files** ip1 ip2
Code Files ip.cc
}

7.3 Refine the Base

Some advanced features of YAKL can be used in a second step to refine criteria, composite bodies, argument I-O relations, effects, etc. This section lists such features.

7.3.1 Optional Operators in Sequences

In sequential decomposition, some of the sub-operators may be optional, that is depending on the data (usually), their execution in the sequence is possible but not always necessary. They are indicated in the body of their composite parent operator inside brackets **[]**, as in

Body
op1 **-** **[** op2 **]** **-** op3

An optional sub-operator may appear anywhere in a sequence (first or last place also). The program supervision engine will decide *dynamically* whether to apply or not an optional sub-operator. For that purpose, the expert must provide it with optionality rules. They are attached to a composite operator with a sequential decomposition type and they describe whether an optional sub-operator should be applied depending on the dynamic state of the current data, on domain object contents, etc. There must be one set of rule for each optional operator in a sequence. Optionality criteria appear after the body of the parent operator, like the choice criteria. Here is an example:

Optional Criteria for op2
Rule { **name** opt_op2
If
param == 4 ,
image_in.format == f2
Then use_optional_operator op2
}

Note that since this rule is in the *parent operator*, `param` and `image_in` are input arguments (parameter and data in the example) of the parent operator, not of the optional child!

An operator used as an optional sub-operator in a body **must** have the same number of input data and of output data (extra data are allowed if they are at the end) in the same order and of the same type. This is compulsory since when the operator is not executed, because the optional criteria does not apply, the input data are directly passed to the output data by the engine. This data flow is managed by the engine and must not be expressed in the **Flow** part.

Remember:

- Optional operators must have the same number, order and types of arguments in and out.
- No extra data flow to write.
- Optionality rules are located in the parent sequence operator.

See pages 68 and 73 for syntax details.

7.3.2 More on I-O Relations

By default output data description are empty, unless something has been done by rules or in **I-O Relations**. In order to express that the output description of some argument is the same as the description of an input one, the idea is first to do a global copy using an equal sign = (ex: `Out1` = `In1`). Note that this is of course only possible if the data have the same type! Then you can assign (using **:=**) some attributes (only those which have to change) .

I-O relations are a way of expressing some semantics about what has been done by the program on output data. (*E.g.,* for a type derived from `Image` the I-O relations may express that from an image of pixels as input an operator has produced as output an image of segments, that the output image size has been reduced compared to the input, that the noise has been removed, etc.).

Remember:

- No transfer of information is done by default from input data to output data descriptions.
- I-O relations may express the semantics about the job of the operator.

See page 62 for syntax details.

7.3.3 More on Alternative Decompositions

It should be noted that alternative decompositions correspond to exclusive choices. If one wants to execute both sub-operators in some cases, you have to define a more complex architecture, described in figure 7.2:

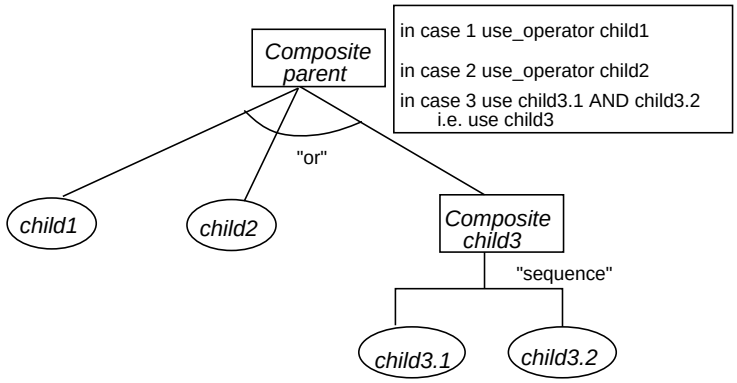


Figure 7.2: Architecture to define non exclusive "or". An abstract version of the choice rules is indicated.

7.3.4 Effects

Effects are another way (along with semantical information in argument types and I-O relations) to express what the represented program does on its output data. The effects and (pre/post)conditions allow the program supervision engine to determine if an operator is applicable in a state or which operator is suitable to reach a given goal. Contrary to the I-O relations (where are expressed relations that depend on the inputs), effects impact on the outputs only (it is of no use to modify inputs *after* their utilisation!).

Effects

out_image.shape_situation := isolated_disconnected

Like pre- and post-conditions effects are separated by commas. In this example, `out_image` must be of a type that has an attribute named `shape_situation` of symbolic type itself,

and the symbolic value `isolated_disconnected` must be allowed for this attribute, *i.e.* it belongs to its range, or there is no range (in this case the value is free, but the expert must be cautious in using it somewhere else!). The effects assign values that can be checked afterwards in preconditions or rule premisses, for instance. Let us assume that the attribute `out_image` is connected to the attribute `in_image` of a (alternative) brother operator in a sequence. In this brother operator the symbolic value `isolated_disconnected` may then be tested, for example in a choice rule like:

Choice criteria
Rule { name choice1
If in_image._shape_situation := isolated_disconnected
Then use_operator op_for_isolated
}

7.3.5 Attribute Value from a File

When results are collected in a file they can be accessed in the **Effects** part, by the **item** notation. The file name is either denoted directly or it may be computed by a method returning a string. The place to search for the interesting information is indicated by a number (an integer), which is the index of a word in the whole file or in a line (sequence of words, terminated by a newline). The precise line to search in may be specified by any significant word appearing in it (like xmin in the following example). It should be noted that for the index computation *delimiters* (*i.e.* space tab ; : =) don't count as words.

Also note that effects are executed even in simulation mode (so files containing the proper information must exist).

Effects
ix := **item** 2 (Integer) **in** "file.r" xmin ,
iy := (1 + (ie.y_size / 2)) - (y / 2)

If the file `file.r` contains the line:

```
xmin = 3
```

`x` will be assigned the value 3, which is considered as the *second* word in the line containing `xmin` (since = does not count). If the item cannot be found before the end of the line, the engine stops with an error message.

The result is always a simple value (int, float or string) and the expert must mention the type the result must be converted to (like `(Integer)` in the example) in order for the engine to type the information extracted from the file.

Remember:

- "Cast" the extracted value, using (Integer) or (String) or (Float).
- Delimiters don't count.

See page 65 for syntax details.

7.3.6 Assessment Criteria

The assessment rules are the most difficult to define, they usually come late in the KB development process. They apply on *output data* of an operator.

Assessment criteria
Rule {
name retry
If Out1.number >=1
Then assess_operator pb_retry repair
}

The "repair" symbol in the conclusion of the rule means that the result is too bad to continue, and that the repair mechanism must be triggered. The attribute `number` must have been defined in the type of the argument `Out1` and must have been filled either by a rule, or by and I-O relation. The `pb_retry` symbol is given by the expert and constitutes a "judgement" which associated by this rule to the current operator. This "judgement" can be searched for by other rules later in the reasoning (usually repair rules that decide what to do depending on the judgements). That is why the symbols used in assessment rules as well as in repair rules must be consistent,

In assessment criteria, like in initialisation ones, there may be a chaining of rules which conclude to a bad (or good) assessment. The assessment may also be left to the user, in this case, the expert proposes a list of values and possibly a question, like in the following example ("Regularity, number and size of the shape?"). The end-user will only have to select one assessment in the list `(round_regular_shape irregular_shape multiple too_small ...)`.

Assessment criteria
Rule { name eval_segmented_im
comment "regularity criteria"
If true
Then display segmented_im,
assess_data_by_user segmented_im
[round_regular_shape irregular_shape multiple too_small ...]
"Regularity, number and size of the shape?"
}
Rule { name eval_segm_morpho_bin_1
If assess_data? segmented_im round_regular_shape
Then assess_operator good continue
}
Rule { name eval_segm_morpho_bin_2
comment "quality is not so good"
If assess_data? segmented_im irregular_shape
Then assess_operator poor_quality repair
}
.....

Remember:

- No repair will be triggered without a bad assessment (*i.e.* "repair" in an action of an assessment rule).
- The expert can leave the assessment to the end-user.
- The symbols used for assessments must be consistent in all rules
- Assessments are difficult at low-level, work bottom-up.

See pages 71 and 75 for syntax details.

7.3.7 Repair Strategy

Two kinds of rules are involved in the failure-handling or repair strategy: *adjustment* and *repair* rules. The former are local to an operator and are triggered by a `re_execute` action, while the latter are more global and used to propagate problems from the point (operator) where they can be detected to the point (operator) where they can be solved. Repair criteria express judgement propagation in a knowledge base for "reactive execution".

Reparation rules (*i.e.* both repair and adjustment rules) are triggered *only if* some assessment rule has concluded on the necessity of repairing. Repairing usually means first, to backtrack to some previous state where the origin of the currently detected problem might be (bad choice or bad parameter value) and second, to start a new line of reasoning with different options (*i.e.* new choice or new parameter value).

Backtracking is possible because PEGASE+ manages a state tree (or dag). States store information about the successive situations during the reasoning of the KBS. They reflect changes in data in particular via transitions that memorise each slot modification of objects.

A state tree (or dag, i.e. a tree without cycles) describes the successive states of a problem solving process. Each state, except the root of the tree, has one or several ancestor(s) and one or several successor(s), except for leaves which have no successors. Starting from the root initial state the system expands it into its successors, each one being obtained by applying a (set of) operator(s) that modify the fact base. Operators applied to go from one state to another are selected with respect to their level of interest to achieve the targeted goal. At anytime, there is only one current state. After or during reasoning it may be necessary to backtrack to a previous state to avoid impasses or to inspect other states. A state may be reversible or not depending if the transitions that led to it can or cannot be "undone"; default is reversible.

Local Repair: Adjustment Criteria

A frequent local repair strategy is simply to re-execute the current operator with modified parameter values. Such a re-execution follows a `re_execute` action like in:

Repair criteria
Rule { name rex
If assess_operator? op1 pb
Then re_execute
}

After that kind of rule, adjustment is trigerred. Note that, to be adjustable, a parameter **must** have a range. Let us assume that `file_in` is of type `MyFile`, and `param` of type `Integer`:

Adjustment criteria
Rule { name adj1
If file_in.number == 1
Then param := 3
}
Rule { name adj2
If file_in.number == 2
Then param := 9
}

Then, if `param` is used in the calling syntax of its operator, like:
syntax program -option *param* **endsyntax**
the first call will generate `program -option 3`, while the second one will lead to `program -option 9`.

Example of an adjustment rule referring to the current operator assessment (which has been set in another adjustment rule, previously in the execution):

Adjustment criteria
Rule { name r_adjust
If
assess_operator? ambiguous
Then
decrease param *decrease the parameter value of 1 (by default)*
}

Assessment on arguments play an important role in the repair process. Here follows another example of adjustment rules in image processing, referring to an argument assessment and assigning the adjustment method of a parameter. In this example, after a primitive extraction on stereo images, an evaluation rule can assess that the number of matched pairs of primitives is insufficient; in such a case, it is possible to adjust the input parameters of the corresponding operator `stereo_match` which are a threshold on the magnitude of the gradient (`thr_m`), and a threshold on the orientation of the gradient (`thr_o`). Below are shown two adjustment rules for this operator. The first one states that the adjustment method for the parameter (`thr-o`) is a method by percentage. Associated methods are predefined for integers (`percent_integer`) and for floats (`percent_float`). Note that the adjustment methods *demand* the adjusted parameter to have a range defined! Otherwise no adjustment is done.

Adjustment criteria
Rule { name radjust
If ....
Then adjustment_method thr_o percent_float,
adjustment_step thr_o := 0.3 **}**

The second rule states that if the number of matched pairs of primitives is insufficient, the two parameters `thr_m` and `thr-o` have to be increased:

If assessed_parameter? number_primitives insufficient
Then increase thr_m ,
increase thr_o

Remember:

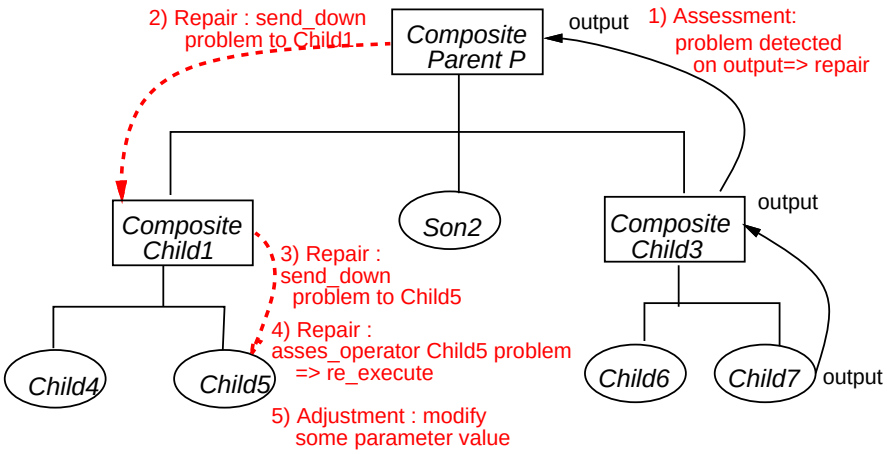
- One problem may lead to modify several parameters.
- Each problem propagation "chain" must end on a primitive operator re-execution, after adjustment of some of its parameters.
- Don't forget to give a range to parameters that may be adjusted.

See page 77 for syntax details.

Global Repair Criteria

In global repair rules the expert expresses information propagation, which means where a bad evaluation information should be transmitted to. Actions in these rules are `send_operator`, `send_up`, `send_down`. They allow the expert to express a strategy of repair and information propagation in a complex hierarchy of operators. For instance, the expert can express that the bad evaluation information has to be transmitted to a sub-operator (`send_down`), or to the parent operator (`send_up`), or to any operator previously applied (`send_operator`). Note that `send_down` can only be used in a composite operator which knows its children.

All these actions lead to some backtracking (see figure 7.3). Should backtrack occur, data and domain object recover their previous values, but it is not the same for parameters, that keep their current value (that is why several adjustments are possible). For instance, if in state 2 an integer input data of Son1 is equal to 3, and if in state 4, it is equal to 6, when the system backtracks to state 2, the value is 3 again.



Evolution of the plan:

State 1 : plan = ... P	P not yet decomposed
State 2 : plan = ... (P) Child1 Child2 Child3	decomposition of P
.....	and so on...
State n : plan = ... (P) (Child1) ... (Child2) (Child3) Child4 Child5	decomposition & execution of children 1 & 2 and decomposition of Child3
State n+1 plan = ... (P) (Child1) ...(Child2) (Child3) (Child4) (Child5)	execution of children 4 & 5 problem detected
Start a new line of reasoning	
State 2 : plan = ... (P) Child1 Child2 Child3	bactrack to state 2 (where Child1 was not yet decomposed)
.....	execution of new line continues
State m : plan = ... (P) (Child1)... (Child2) (Child3) Child4 Child5	re−execution leading to a new state m

Figure 7.3: A "repair chain". The operators indicated in parenthesis in a plan are executed or decomposed at the current step of reasoning.

Repair criteria
Rule {
name r1
comment "If the output is ambiguous, raise problem to child S1"
If assess_operator? F ambiguous
Then send_down S1 output_ambiguous **}**

This first rule is associated with the parent operator (`P`) it detects a problem and sends down a name of problem to one of its children. The name of the problem is just a symbol (preferably meaningful). The only constraint is that the same name must be used in all rules referring to the same problem (*i.e.* use same names of problem in send-* and assess-*). But a problem named *e.g.,* "output_ambiguous" at one level may raise a problem named differently (*e.g.,* "ambiguity") at another level (see below).

The same type of rule is used by `S1`:

Repair criteria
Rule {
name rdown
comment "If ambiguous, suspect S5"
If assess_operator? S1 ambiguous
Then send_down S5 ambiguity **}**

This second rule propagates the problem diagnosis to another child operator `S5`, which is suspected to be the origin of the problem. In this operator a repair rule triggers the re-execution, and adjustment rules modify some of the values of parameter of `S5`, before its re-execution.

Repair criteria
Rule {
name rex
If assess_operator? S5 ambiguity
Then re_execute **}**

Assessment criteria
Rule {
name rass
If ....
Then increase param **step** 3
}

Remark: It is rather bad programming to use `send_up` if you can avoid it, because it violates the encapsulation principle (*i.e.* an operator is a reusable black box). Using `send_up` assumes that an operator knows that it has a parent. Moreover, if the operator is used more than once (it has several parents) this implies that *all* its parents know how to fix the problem. Instead, it is better to evaluate the results at the parent's level and to propagate the potential problems to children (a parent knows that it has children, there is no encapsulation violation). It is often the case that the evaluation and the corresponding send_up are attached to the last operator of a sequence, the results of which are also the results of the parent operator, so the evaluation can be done at the parent level.

Remember:

- Be consistent in naming the problems in rules that raise them, and in rules that catch or transmit them.
- Prefer top-down propagation (from a parent to its children)

See page 76 for syntax details.

7.3.8 More on Assessment

Assessment criteria not only allow to check properties of output of judgement of data or operators, but also to check how many times an operator has been re-executed. For example, `nb_previous_assess` return the number of times an operator has been re-executed due to a given problem. Here is an example of how to use this action (see page 71, for other advanced assessment actions). One parent with 2 children op1 and op2, tries to repair twice the same problem (by a re-execution of op1 with different parameter values) but resigns after:

```
Assessment criteria
Rule {
        name retry1
        If Out1.size >=1 ,
              nb_previous_assess op1 pb <= 2
        Then assess_operator pb repair
}
Rule {
        name  retry2
        If Out1.size >=1 ,
              nb_previous_assess op1 pb > 2
        Then printf ("No solution, stop");
}
```

7.3.9 Parameter Flow

A "Parameter Flow" section may come after the **Flow** part. It allows a parent to enforce the values of parameters of its children.

```
Parameter Flow
        erosion.extension := ".det"
```

For the moment, only assignments to fixed values (like in the example) are allowed, because a lot of problems are still open, *e.g.*, "Should the values of parent and child parameters be shared after such assignments?"

7.3.10 Methods for Expert Types

The expert may also redefine methods (for instance `display` is defined for `Image` and may be redefined in sub-types), or define new ones (`compute` in the following example). In the type description only the method signature is needed. The code itself has to be written in the engine language (C++ or Lisp) in so-called "code files" (see page 104).

Argument Type {
    name MyImage
    Subtype Of Image
    Attributes
    .....
Methods void display ()
    Integer compute()
}

For the moment, only methods with no argument are accepted. No argument means no other argument than the implicit one *i.e.* the instance on which the method applies (an instance of `MyImage` for the method `display` in the example).

Chapter 8

Detailed Example of a Simple Knowledge Base

As an example, this chapter describes the almost complete description of a simple knowledge base, divided into several files (`.yakl`). The base corresponds to the processing of satellite images which can be of two types (either Spot or Landsat).

Remember:

- Define as many `.yakl` files as necessary; they may "import" each other.
- Describe once your KB (in a *separate* `.kb` file),
- Describe *at least one* Functionality,
- Associate the functionality with *at least one* operator -in both directions (a Functionality *achieves* some operator(s) and an Operator has an attached Functionality).

8.1 KB definition

First, a file named `kb-cocktail.kb` contains the knowledge base description *alone*:

Kb {
    name cocktail
    Complete Name "Satellite images Image Processing"
    Authors "X Y"

```
    Kb Path "/usr/local/lama/bases/Cocktail" "
    Version 2.0
    Root Node cocktail
    List of Files types prim comp req
}
```

The list of files contains the names of the `.yakl` file that constitute the KB, without the extension. These files are described in the following.

8.2 Type and Domain Object Definitions

Second, in a file named `types.yakl` are the argument types and domain objects:

```
Argument Type {
    name Spot
    Subtype Of Image
Attributes
    String name path      # overriding path attribute
        default "/usr/local/Tests/SPOT/"
    String name extension    # overriding extension attribute
        default ".spot"
Methods Void display()
}

Object Type {
    name Context
Attributes
    Integer name Nb_spectres_used
        default 3
}

Object Instance {
    Context name context
}
Argument Instance {
    Spot name image1
Attributes
    basename := "spot1"
}
# And the same for 2 other instances: image2 and image3
...
```

The `Spot` type is defined by the expert. It will be used for some operator's arguments. Its instances `image1..3` will be used in a request.

8.3 Operators

A `prim.yakl` file gathers the descriptions of the primitive operators. It imports the `types.yakl` file, because the `Spot` type is necessary for typing some data arguments.

```
Import types
Primitive Operator{
      name Varimax
      Characteristics threshold hysteresis
      Input Data
            Spot name xs1                          use of Spot type, expert-defined
            Spot name xs2
            Spot name xs3
      Input Parameters
            Integer name list
                  default 0
                  range [ 0 2 ]
            Float name w
                  range [ 0.0 2.0 ]
                  default 0.0
      Output Data
            Image name fact1
            I-O relations                          for output image, versus input one
                  fact1.path := xs1.path ,         it is at the same place
                  fact1.basename = xs1.basename,   it has the same basename
                  fact1.extension :=".fact1"       but a different extension
....
Preconditions
            valid xs1 ,
            valid xs2,
            valid xs3
Postconditions
            valid fact1,
            ..
Assessment criteria
Rule {
      name eval1
      If true
      Then
            assess_data_by_user fact1 [incorrect correct]
                                                   only the user can do that
      }
Rule {
```

```
        name eval2
        If assess_data? fact1 incorrect
        Then  assess_operator_by_user [bad ok] [repair continue]
        }
Rule {
        name eval3
        If assess_data? fact1 correct
        Then  assess_operator ok continue                    no problem
        }
Initialization criteria
Rule {
        name rinit1
        If true
        Then  w := 2.0
}
Rule {
        name rinit2
        If true
        Then  list <- [0 1 2]
}
Adjustment criteria
Rule {
        name radjust1hafq3XS
        If true
        Then  increase w ,
              increase list step 2
}
Call
        language shell
        syntax Varimax -XS1 xs1.get_filename() -XS2 xs2.get_filename()
              -XS3 xs3.get_filename() -list list -w w
        endsyntax
        type real
}
...
```

Note: If xs1 corresponds to a file named `F1.spot`, xs2 to a file named `F2.spot`, xs3 to a file named `F3.spot`, list equals 1, and w equals 0.5, then the calling sysntax will generate the following command:

```
Varimax -XS1 F1.spot -XS2 F2.spot -XS3 F3.spot -list 1 -w 0.5
```

All the primitive operators are described in the same way.

Then, a `comp.yakl` file corresponds to composite operators. One of them is the entry point to the KB: `spot_operator`. It is the only one (in this base) which is associated with

a functionality. Its abstract functionality `Spot_processing` is described in the same file, because they must know each other. It imports the `types.yakl` and `prim` files, because it uses the `Spot` type in argument description and primitive operators in composite operators' bodies.

```
Import types prim
Functionality {
    name Spot_processing
Achieved by by spot_operator
Input Data
    Spot name xs1
    Spot name xs2
    Spot name xs3
Output Data
    Image name img1
    Image name img2
}

Composite Operator {
    name PointsChoice
    Input Data
        Image name histo
Output Data
    File name out
    Preconditions
        valid histo
Postconditions
        valid out
    Body
        Triangulation | RectConvPtIn
Choice criteria
Rule {
    name Choice3Points
    Let c a Context
    If c.Nb_spectrum == 3
    Then use_operator Triangulation
    }
Rule {
    name Choice4Points
    Let c a Context
    If c.Nb_spectrum == 4
    Then use_operator RectConvPtIn
    }
```

```
Rule {
        name Choice5Points
        Let c a Context
        If c.Nb_spectrum == 5
        Then {{printf( "Not yet implemented ");}}
        }
        Distribution
                PointsChoice.histo / Triangulation.histo
                PointsChoice.out / Triangulation.io
                PointsChoice.histo / RectConvPtIn.histo
                PointsChoice.out / RectConvPtIn.out
}
...

Composite Operator {
        name rotation
Input Data
        Spot name xs1
        Spot name xs2
        Spot name xs3
Output Data
        Image name fact1_o1
        Image name fact2_o1
        Image name fact3
        Image name frq
Preconditions
Postconditions
        ...
        Body  Varimax - Octet
        Distribution
                rotation.xs1 / Varimax.xs1
                ...
        Flow
                Varimax.fact1 / Octet.fact1
                ...
}
...
Composite Operator {
        name spot_operator
        comment "highest level operator for Spot images"
Functionality Spot_processing     Entry point for requests
Input Data
        Spot name xs1
```

Spot name xs2
Spot name xs3
Output Data
Image name img1
Image name img2
Preconditions
...
Postconditions
...
Body
rotation - HistoGris - Triangulation - LocalSpectr - DiagTri2D - Melang2DClassEntrop
Distribution
PointsChoice.histo / Triangulation.histo
PointsChoice.out / Triangulation.io
spot_operator.xs1 / rotation.xs1
spo_operator.xs2 / rotation.xs2
spot_operator.xs3 / rotation.xs3
spot_operator.img1 / Melang2DClassEntrop.img1
spot_operator.img2 / Melang2DClassEntrop.img2
Flow
rotation.fact1_o1 / HistoGris.img1
...
}

8.4 Request

Finally, the `req.yakl` file contains a request referring to the `Spot_processing` functionality. The request uses the previously defined instances `image1..3` as attribute values. It imports the `types.yakl` file because it is where these instances of `Spot` images are declared and the `prim.yakl` file, because it is where the functionality is declared.

Import types prim
Request {
Spot_processing **name** RSpot
Attributes
xs1 := image1
xs2 := image2
xs3 := image3
}

Chapter 9

Graphic Interface

GIPSE (Graphic Interface for Program Supervision Engines) is a graphic interface for manipulating and executing knowledge bases in program supervision.

This document is a guide to the GIPSE interface. It addresses mostly experts either building a new knowledge base, or modifying/auditing an existing one through visualisation. All examples shown come from the Progal knowledge base. Figure 9.1 is a picture of GIPSE's main window with the Progal knowledge base loaded.

9.1 General Overview of GIPSE

This section gives a brief description of GIPSE and its utilization. All features are fully developped in the following sections.

9.1.1 On Line Help

Located on the upper righthand corner of the main window, the help button (see figure 9.2) gives online help on commands. To get information on a command, click the help button, then click the command itself. For example, clicking the help button and then the zoom button pops up a help window that gives information on the zoom command. (see figure 9.3).

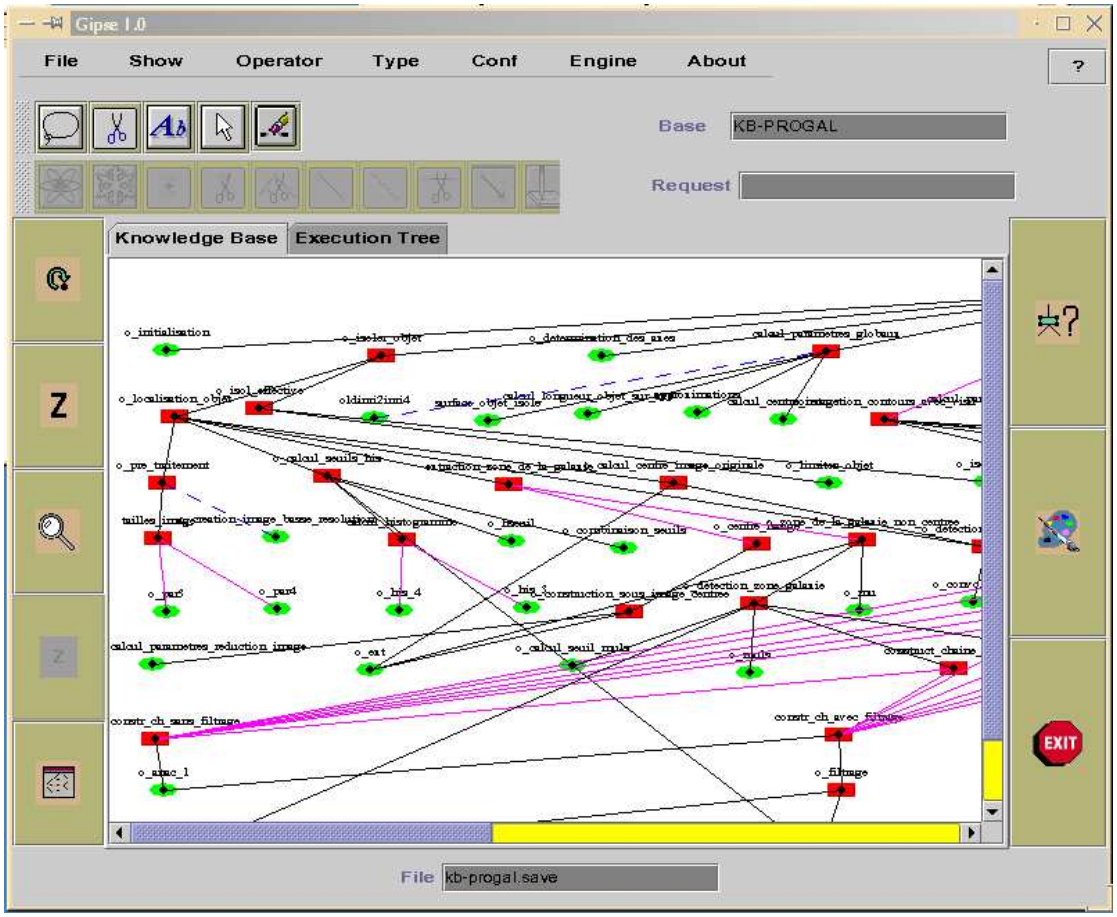


Figure 9.1: GIPSE's main window

Figure 9.2: Help button

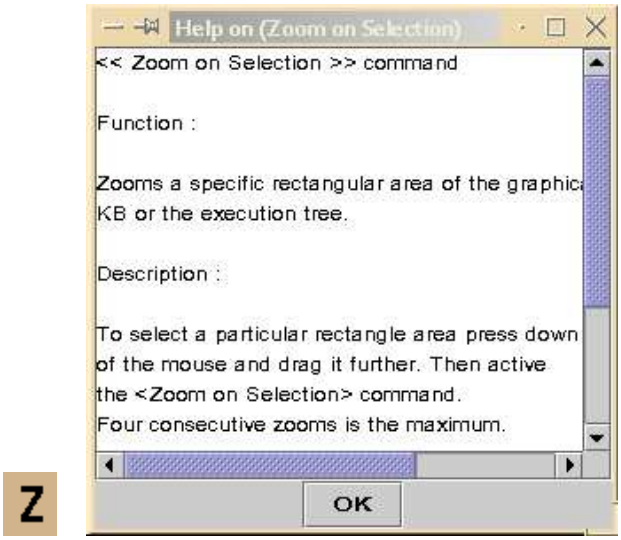


Figure 9.3: On the left, zoom button, on the right, help window for the zoom command

9.1.2 Opening a Knowledge Base

Knowledge bases are organized in `.yakl` files located in the directory wearing the knowledge base name. Furthermore, some important information is stored in `.save` files used by the GUI. YAKL files contain the expert knowledge while `.save` files are automatically generated from `.yakl` files. For example the directory `KB-PROGAL` contains all `.save` and `.yakl` files storing the Progal knowledge base.

Figure 9.4: Root kb directory with knowledge bases sub-directories

To open a knowledge base, click the *Open* command in the *File* menu on the menu bar (see figure 9.5).

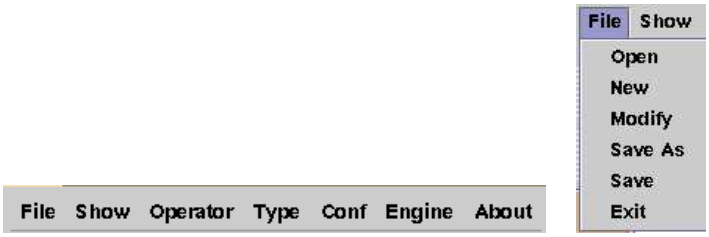


Figure 9.5: The Menu Bar (left) and the File Menu (right)

A file browser opens up showing the content of the knowledge base root directory (see figure 9.6).

Select a knowledge base by clicking on its name, then click the *Update* button to get a list of its files. One of the `.save` file has the same name in lower case characters as its containing directory; this file is the `ROOT FILE` for the knowledge base. Click *OK* to open the corresponding knowledge base. A graphical representation of the knowledge base shows up in the central *Knowledge Base* tab of the main window. Also knowledge base's name appears in the area so labeled in the main window(9.1). See below for graph vs.tree options.

For example, selecting KB-PROGAL as sub-directory then kb-progal.save as root file opens the Progal knowledge base in the main window.

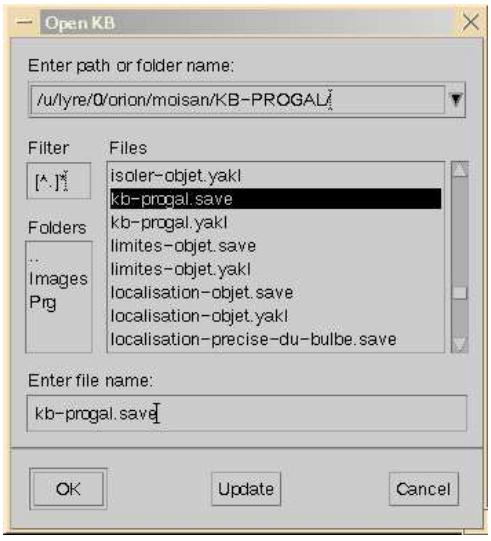


Figure 9.6: File browser for opening a knowledge base

9.1.3 Visualisation of a Knowledge Base : tree vs. graph

A knowledge base can be viewed as an operator's graph showing shared nodes. Labeled colored nodes are operators; colored links represent separate relations of composite operators. Red rectangular nodes note composite operators while green oval ones represent primitives (see figure 9.7).

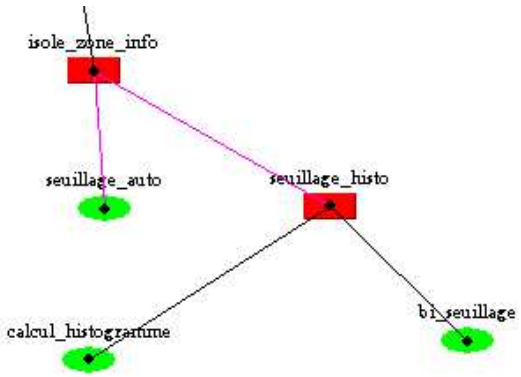


Figure 9.7: Composite operators and their sons

Links are of three types: sequential (black), choice (magenta) and optional (doted blue).

A knowledge base can also be viewed as a tree which is easier for the human eye to follow.(see figure 9.8).

A small black diamond next to the rectangle representing an operator indicates it has sons ; an operator with more than one level of sons heads a list of their names. Use the *Graph-Tree* command in the *Show* menu to switch from tree to graph and vice versa.

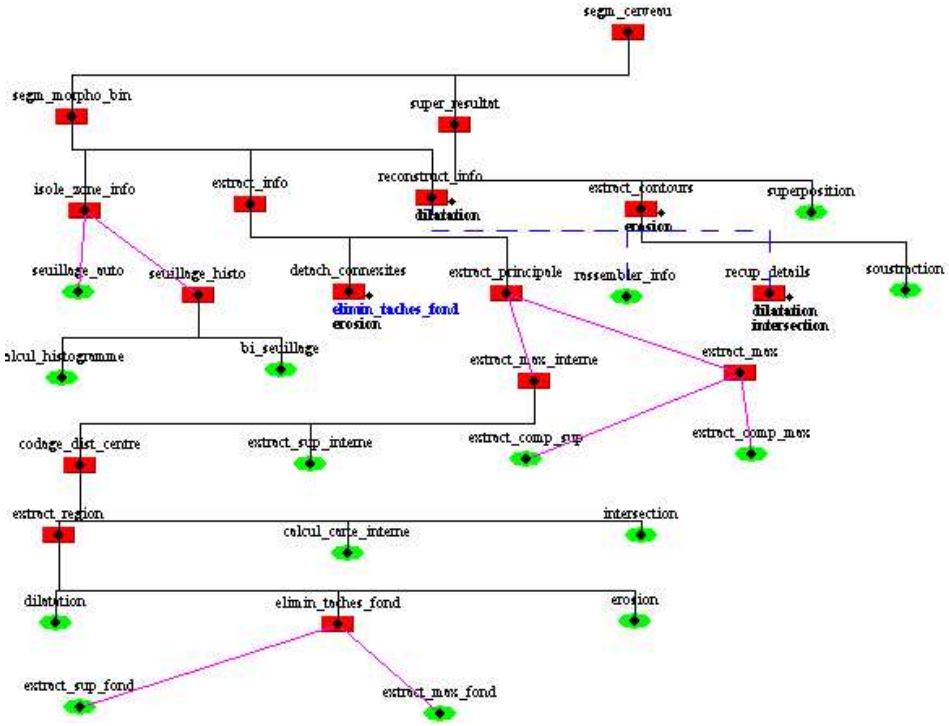


Figure 9.8: Progal knowledge base represented as a tree (part of)

9.1.4 Executing a Request

A request can only be executed on a **fully** loaded base. Use the *Select Request* in the *Engine* menu to do that. A window pops up allowing either selection of a request or construction of one (which is detailed later).

Debug Level and Type of Execution

Debug level is set at 0 unless expert changes it by clicking the appropriate buttons. Choices are :

- Level 0 : running mode with no log
- Level 1 : running mode with some log
- Level 2 : running mode with full log
- Level 3 : running mode with full log and display of rules' YAKL code

Figures 9.9, 9.10 and 9.11 show the three mobile windows associated with an execution.

The expert can also choose a step by step running mode by activating the "yes" option on the button "step by step". When all choices have been completed, start the execution of the request by clicking the *Go* button.

On levels 0 and 1, the *Continue* and *Stop* buttons in the *Control* panel are disabled. The panel's color is red. A primitive operator in the execution plan can be selected at any time for complementary information using the *Info* button.

On levels 2 and 3, the *Continue* and *Stop* buttons in the *Control* panel are enabled when the colored panel turns green. Use the *Continue* button to get to the next execution step.

As soon as execution starts, a new window pops up on which the generated solution plan will be dynamically displayed. Depending on the debug level, a log window can also pop up. Finally, the Execution Tree will be dynamically constructed under the *Execution Tree* tab of the main window.

The engine communicates with the interface via dialog boxes that allow the expert to feed it information while the execution is taking place.

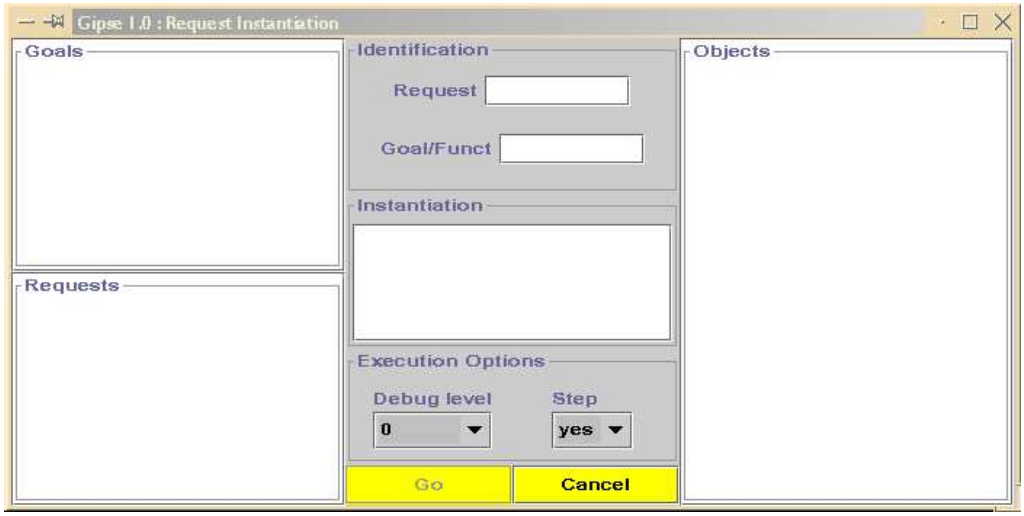


Figure 9.9: Selecting a request

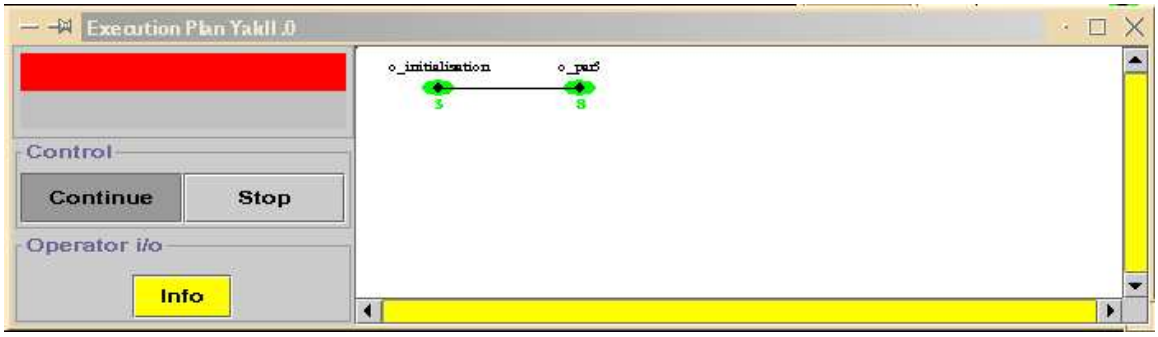


Figure 9.10: Generated solution

Figure 9.11: Execution logs of trace, if debug level > 0

Figure 9.12 shows the resulting Execution Tree.

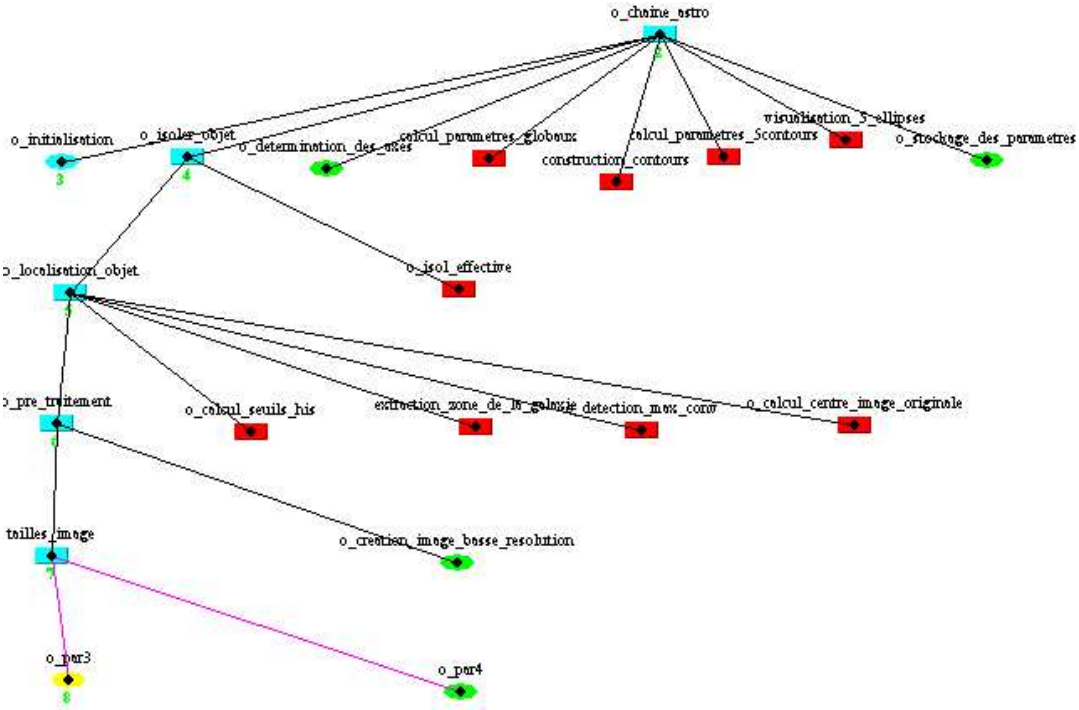


Figure 9.12: Execution Tree

9.1.5 Exiting GIPSE

To exit GIPSE click the *Exit* button on the right vertical bar (see figure 9.13) or use the *Exit* command in the *File* menu.

9.2 Visualisation Tools

9.2.1 Manipulating the Knowledge Base Graph

GIPSE provides tools for manipulating the knowledge base graph.

Figure 9.13: Exit button

Select and Move

Selecting an operator – with the left mouse button – turns its rectangle to grey and enables all commands relevant to this operator like *Show Descriptor* in the *Operator* menu. Selecting a link – with the left mouse button – turns its line to green which may help its visualisation.

Since a complex graph can be difficult to read, the expert may want to use the center mouse button to move operators around : select, press and drag, all with the middle button of the mouse! In order to go back to the initial display, use the *Repaint* command in the *Show* menu or click the repaint button on the right vertical button bar. See Figure (9.14).

Figure 9.14: Repaint button

Zooming, Unzooming

Commands connected to zooming are in the left vertical buttons bar (see figure 9.15) as well as in *Show* menu.

The *Zoom* command (left bar –see figure 9.3– or in *Show* menu) zooms the graph. The maximum number of consecutive zooms is 4. The *Unzoom* command (left bar –see figure 9.15– or *Show* menu) undoes what the *Zoom* command does.

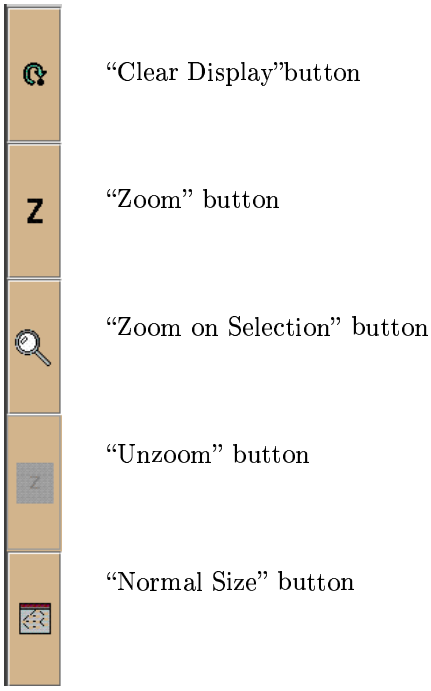


Figure 9.15: Left vertical buttons bar

Point to an area for "zooming on selection" by drawing a perimeter around the chosen area with the right mouse button: a rectangle shows up dynamically. (left bar –see figure 9.15– or *Show* menu).

In order to retrieve the original size after zooming several times without using the "Unzoom" command, use the *Normal Size* command (left bar –see figure 9.15— or *Show* menu)

When working on a complex graph, separate operators along the X or Y axis by activating the *Increase X Spacing-Decrease X Spacing* and *Increase Y Spacing-Decrease Y Spacing* commands in the *Show* menu.

Graph Versus Tree

See also section 9.1.3

Links between sequential operators and their sons are displayed as forks (see figure 9.16).

9.2.2 Getting Information on Operators

The *Operator* menu has all commands relevant to operators.

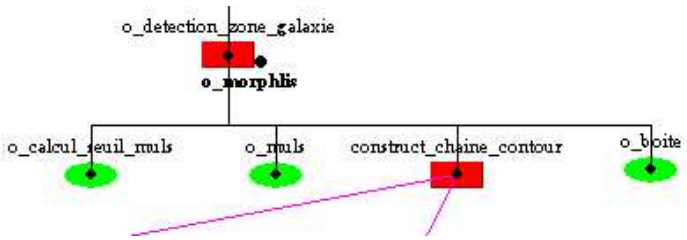


Figure 9.16: Sequential fork link tree representation

Click the *Find* command in the operator's menu or the *Find* button of the right vertical bar to locate an operator in the graph. (see figure 9.17).

Figure 9.17: Find button

A dialog window pops up: enter the desired operator's name. If found, the operator will be centered as best as possible in the window and its rectangle highlighted in grey. On a selected operator, the expert may obtain additional data by selecting "Show Descriptor". A window opens up with fives tabs labelled *Input*, *Output*, *Parameters*, *Rules* and *Yakl* (see figures 9.18, 9.19, 9.20).

- The *Input* tab has a list of the operator's input data and the *Output* tab a list of the operator's output data. The name of the operator is shown at the bottom of the window and every variable can be selected in order to get its type. Clicking the *Type* button pops up a window that give information on the variables' types (see figure 9.21).

  The other two tabs show a button bar (see figure 9.22) related to data flow visualisation.

  A data flow is a link between operator's data. For example an output generated by an operator can become the input of the next operator.

  GIPSE is able to display these data flows by drawing an arrow between the two sets of data. See figure 9.23 for an example of data flow.

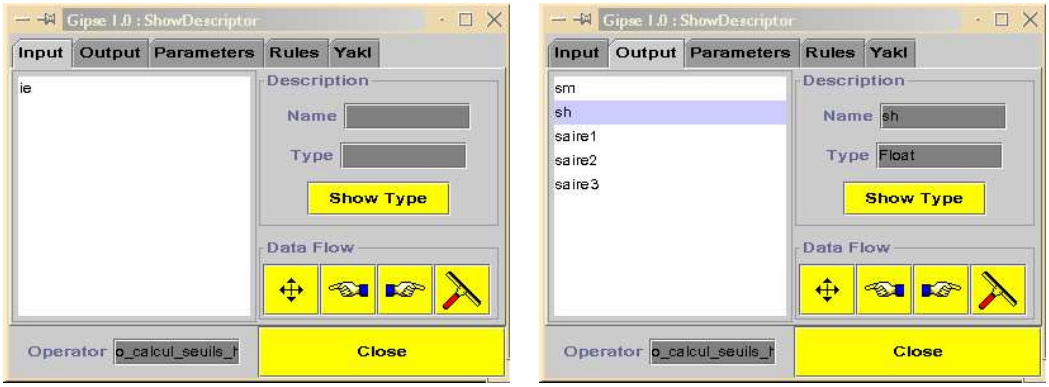


Figure 9.18: Input and Output tabs of Show Descriptor Window

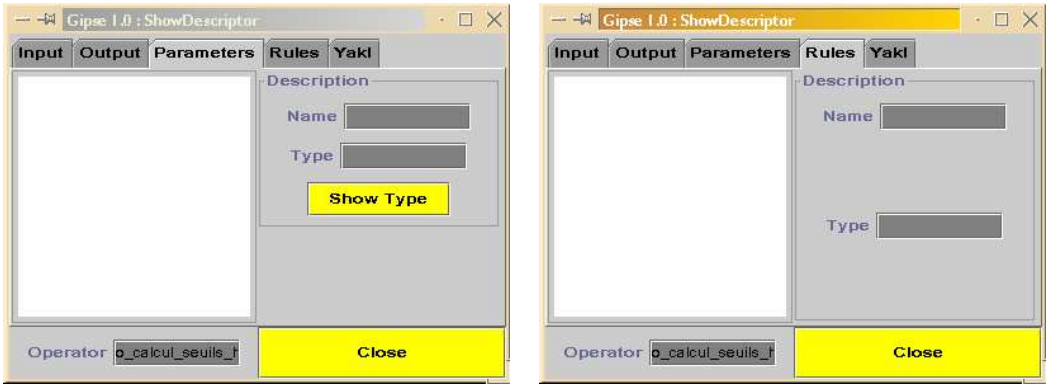


Figure 9.19: Parameters and Rules tabs of Show Descriptor Window

Any operator's data (input or output) shows necessarily some flow since it comes from one or more operator(s) and goes to one or more other operator(s), brother(s), son(s) or father(s) of the initial one(s).

Select data (input or output) to visualize :

- the whole flow going through by clicking the *Whole Flow* button (see figure 9.24)
- the flow that go to other operator(s) by clicking the *Go To* button (see figure 9.25)
- the flow that come from other operator(s) by clicking the *Come from* button (see figure 9.26)

Figure 9.20: Yakl tab of Show Descriptor window

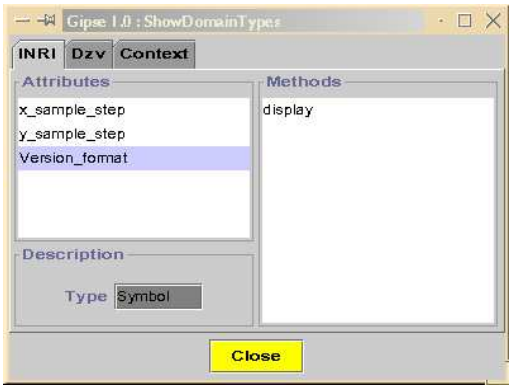


Figure 9.21: Domain types

Figure 9.22: The Data Flow bar

In the last two cases, when flow goes or comes from more than one operator, select one of those operators before clicking again the button to get the flow on this path. Otherwise, GIPSE continues showing the flow from the last displayed arrow flow.

Use the *Clear Data Flow* command in the *Operator* menu or the *Clear Data Flow* button (see figure 9.27) to remove data flows from the graph.

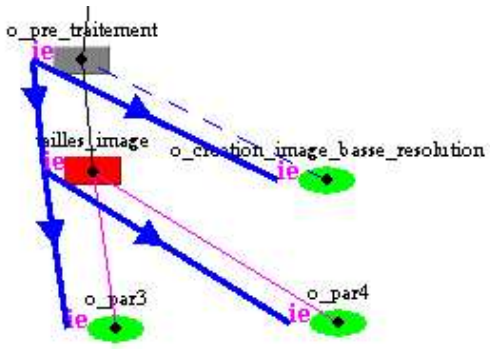


Figure 9.23: Example of data flow

Figure 9.24: Whole Flow button

Figure 9.25: Go To button

Figure 9.26: Come From button

Figure 9.27: Clean button

- The *Parameters* tab shows a list of the operator's parameters, if any. As for the *Input* and *Output* tabs, their types are shown when selected in the list.
- The *Rules* tab shows a list of the operator's rules. As explained earlier, type is shown for the rule selected (types are I (Initialisation), A(Adjustment), C (Choice), R (Repair) or E (Evaluation)).

- The *Yakl* tab shows YAKL code that describes the operator's behavior. Every time you select an item from the lists in the first four tabs (*Input*, *Output*, *Parameters* or *Rules*) the *Yakl* tab shows the corresponding YAKL code.

To get a quick view of the input/output of an operator, use the *Display I/O* command in the *Operator* menu. Input and output are directly displayed on the graph, input to the left, output to the right (see figure 9.28). Use the *Hide I/O* command to undo the *Display I/O* command.

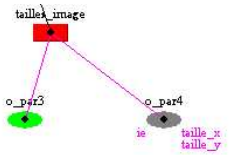


Figure 9.28: Displaying input/output on both sides of an operator

To get a quick view of the type of rules for an operator, use the *Display Rules* command. A string composed of 'I', 'A', 'C', 'R, and 'E' characters will show up below the operator (see figure 9.29). For example an operator which has rule(s) typed 'I' and 'C' will get the *I C* string below it. Use the *Hide Rules* command to undo the *Display Rules* command.

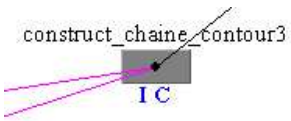


Figure 9.29: Displaying rule types below an operator

9.2.3 Data Types

Data types are of two kinds:

- *built-in*, corresponding to predefined types (integer, float, string,...)
- *domain* types, constructed by the expert.

GIPSE is able to display and load all types.

All commands that are related to types are located in the *Type* menu.

9.2.4 Displaying Types

Use the *Show* item in the *Type* menu to display types. This command yields *Domain* and *Built In* commands. Select *Domain* to get information about domain types, *Built In* for built-in types.

A window pops up that display types (see figure 9.21). Every type has its own tab where it is displayed. Methods and attributs are listed. Select an attribut to get its type.

9.2.5 Loading a Type

Types, like operators, are described in `.yakl` files. Load a domain type by using the *Load-Domain* command in the *Type* menu. Just select a .save file and click *OK*. The domain types stored in that file are loaded.

Note that selecting the root file of the knowledge base loads all domain types related to it.

Using the *Load-Built In* command loads only the YAKL built-in types.

Finally, unload all types by using the *Clear* command.

9.3 Creating and Modifying a Knowledge Base

GIPSE allows the expert to graphically create a new knowledge base or to modify an existing one.

The tool bar shown figure 9.30 is specifically dedicated to these tasks.

To create a new knowledge base, the expert must have already loaded types. When so, use the *New* command in the *File* menu.

To modify an existing knowledge base, use the *Modify* command in the *File* menu which enables the New-Modify bar.

Then use the *New-Modify* bar to do the job.

Figure 9.30: The New Modify knowledge base bar

9.3.1 The New-Modify Bar

A complete description of every button in the bar follows.

- To create and place a new composite operator:

  First press the left mouse button where the new primitive operator is to be displayed then click the Composite button (see figure 9.31.

Figure 9.31: Composite button

A dialog box pops up to enter information on the new operator (see figure 9.32).

First give the operator's name then indicate what type of composite it is (sequence or choice) (see figure 9.32).

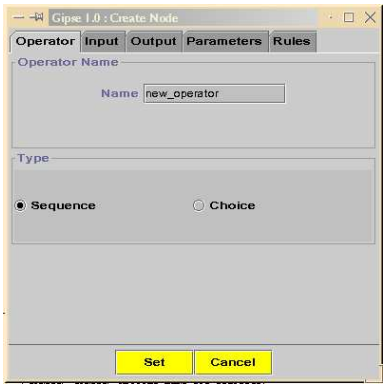


Figure 9.32: Dialog window to set a composite operator

Use the *Input* tab to set input for the operator (see figure 9.33):

- Enter input's name in the text box
- Choose *Built In* or *Domain* type via the radio buttons on the *Choose* panel then use the proper checkbox of the *Types* panel.
- Click the *Add* button (see figure 9.35) to add that input to the operator
- Use the *Remove* button (see figure 9.36) to remove a selected input

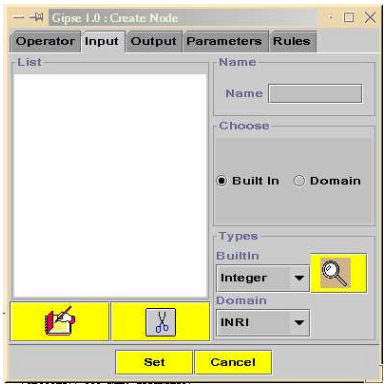


Figure 9.33: Dialog window to set an operator (*Input* tab)

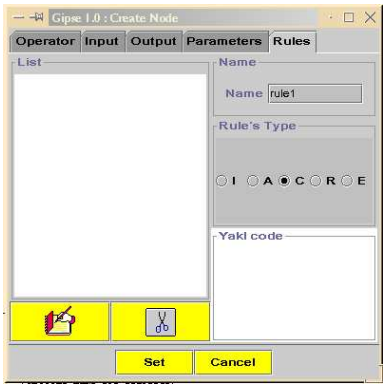


Figure 9.34: Dialog window to set an operator (*Rules* tab)

Figure 9.35: *Add* button

Figure 9.36: *Cut* button

In the same or a similar manner, under *Output*/*Parameters*/*Rules* tab, enter output and parameters to the operator. The *Rules* tab is shown on figure 9.34.

Clicking the *Set* button on this dialog box upon completion.

The composite operator is placed on the graph at the desired location.

- To create and place a new primitive operator, proceed as for a composite operator, clicking the Primitive bar button (see figure 9.37). There is no type to set (sequence, choice) for a primitive operator.

Figure 9.37: Primitive button

- To modify an operator, select it first then click the *Modify Node* button (see figure 9.38).

  A dialog box pops up allowing modification. Click the *Set* button upon completion.

Figure 9.38: Modify Node button

- To remove a selected operator, select it first then click the *Cut Node* button (see figure 9.39).

Figure 9.39: Cut Node button

- To remove a sub tree, first select the root operator of its sub-tree then click the *Cut Tree* button (see figure 9.40).

Figure 9.40: Cut Tree button

- To create a link between two operators:

  First select with the left mouse button the son operator then, holding the button down, drag the mouse towards the new father and release the button upon reaching it. A line is drawn between the son and its father. Click the *Link* button (see figure 9.41). Note that the son's links to its brothers are taken into account.

Figure 9.41: Link button

- To create an optional link between two operators:

  The procedure is the same as above using this time the *Optional Link* button (see figure 9.42) instead of the *Link* one. An optional link instead of a regular one is created.

Figure 9.42: Optional Link button

- To remove a link:

  Select a link then click the *Remove Link* button (see figure 9.43).

Figure 9.43: Cut Link button

- To create data flows between two operators:

  First press the left mouse button while pointing on the first operator then drag the mouse towards the second one and release the button when reaching it. A line is drawn automatically between the two operators. Click the *Manage Data Flow* button (see figure 9.44), a dialog box pops up (see figure 9.45) allowing the expert to create/remove data flows.

  To create a data flow, simply select the two sets of data from the left panel then press the *Add* button (see figure 9.35). To remove a data flow, select it and click the *Remove* button (see figure 9.36) on the right panel.

  Press the *Set* button upon completion.

Figure 9.44: Manage Data Flow button

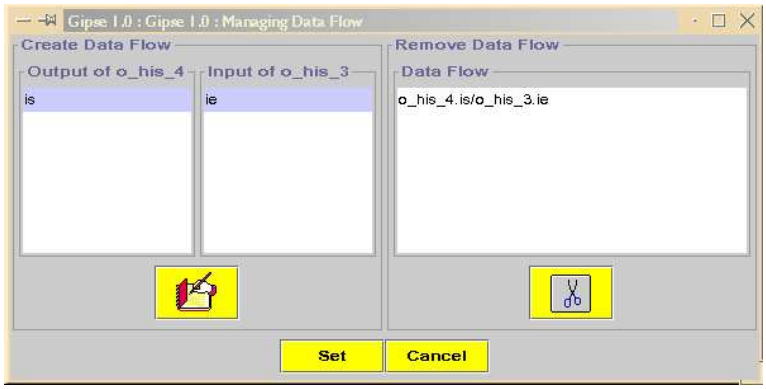


Figure 9.45: Managing Data Flow

- To update the graph after modifications, click the *Draw* button (see figure 9.46).

Figure 9.46: Draw button

9.3.2 Saving a Knowledge Base

A knowledge base is originally distributed in several .save files. GIPSE saves a knowledge base in a unique .save file. To save a new or modified knowledge base, use the *Save* or *Save As* command in the *File* menu. *Save As* allows the expert, via a files browser, to choose the file's name while *Save* uses the last chosen file.

Thus use first *Save As* then *Save* to update modification on the same knowledge base.

Note that the saving process stores not only the operators in the same .save file but also domain types related to the knowledge base.

9.3.3 Creating or Modifying a Knowledge Base Type and Saving It

GIPSE allows the expert to create and modify all domain types. This is done using the *Create* and *Modify* command in the *Type* menu.

The *Modify* command is available only if types have been loaded.

When using the *Modify* command, a dialog window pops up that gives a list of domain types (see figure 9.47).

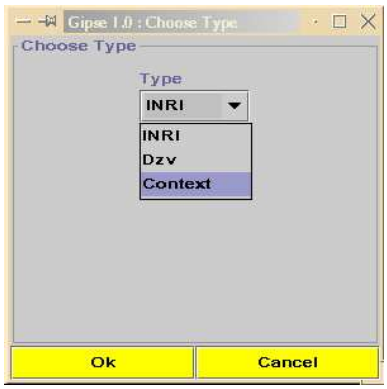


Figure 9.47: List of domain types

Select in that list the type to modify then click the *OK* button.

Then in both cases (modify or create) a dialog window pops up to access type information (see figure 9.48).

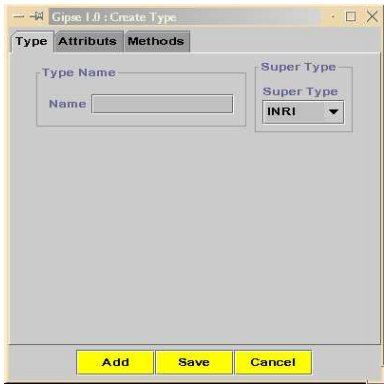


Figure 9.48: Creating a type

Use the *Attributes* and *Methods* tabs to set attributes and methods to the type. These dialog tabs work the same as the ones for setting input, output, parameters and rules for an operator (see figures 9.49, 9.50)

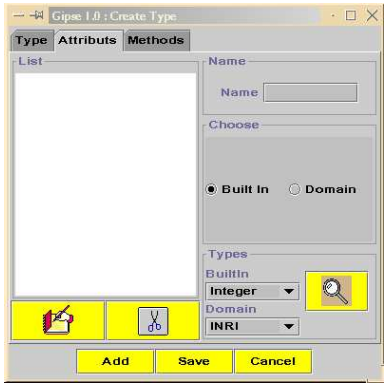


Figure 9.49: Setting attributes to a type

- When modifying a type, click *Set* to save modifications.
- When creating a new type, click the *Add* or *Save* button upon completion. In both cases, a file browser pops up so that the expert selects a .save file for saving. In the *Add* case, the type will be added to the file. In the *Save* case the file will be overwritten by the type.

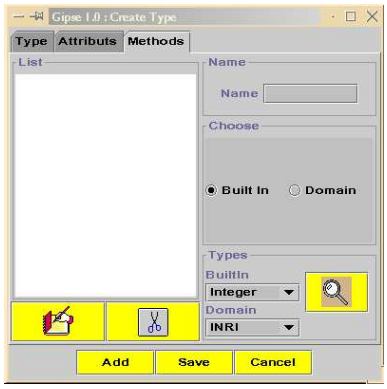


Figure 9.50: Setting methods to a type

9.4 Adding text and arrow on your Knowledge Base

To include in a document a Knowledge Base's graph or its Execution Tree, use any grab tool on your operating system. On Solaris, for example use *snapshot*.

To write personal annotations or draw arrows on a knowledge base's graph or on an executing tree use the graphic features bar (see figure 9.51).

Figure 9.51: The Graphic Features bar

9.4.1 The Graphic Features Bar

A complete description of the function of every button in the bar follows. A *graphic feature* is here either arrow or text.

Figure 9.52: Text button

- To write:

  First click the *Text* button (see figure 9.52) then click the right mouse button where the text is wanted. A dialog box opens up: enter your text! It will be written at the desired place.

Figure 9.53: Arrow button

- To draw an arrow:

  First select the *Arrow* button (see figure 9.53) then press the right mouse button at the origin of the desired arrow. Drag the mouse to construct it. Release the right mouse button to end your arrow.

Figure 9.54: Selection button

- To select a graphic feature, first click the *Select* button (see figure 9.54) then press the left mouse button next to the graphic feature you want to select. It will be surrounded by a dotted rectangle.

Figure 9.55: Cut button

- To remove a selected graphic feature, first select it then click the *Cut* button (see figure 9.55).

Figure 9.56: Clear all button

- To remove all graphic features:

  Simply click the *Clear Features* button (see figure 9.56).

9.4.2 Moving Graphic Features Around

As operators, all graphic features can be moved around by the expert. To do so, use the center mouse button. Click on a graphic feature, hold the center mouse button down while dragging it.

9.5 Execution

GIPSE is linked to a program supervision engine that allows the expert to execute requests on a loaded knowledge base. When a request has been selected, the engine starts the execution. GIPSE displays dynamically the generated solution plan, logs information generated by the engine and draws the Execution Tree while being constructed by the engine.

9.5.1 Selection of a Request

To select a request use the *Select Request* command in the *Engine* menu. A dialog window opens up listing pre-defined knowledge base requests in the *Requests* panel (see figure 9.9).

Select one and click the "Go" button to execute it.

Create your own request following the step-by-step process described below:

- First select a functionality in the *Functionality* panel. The *Instantiation* panel is filled up by the attached functionality and its attributes.
- For at least one attribute:
  - Select an attribute in the *Instantiation* panel. The *Objects* panel is filled up by all objects in the knowledge base belonging to the type of the attribute.
  - Select an object in the *Objects* panel.

then click the *Go* button.

After clicking *Go*, a window shows up on which a graphic representation of the dynamically generated execution plan is displayed (see figure 9.10).

9.5.2 Generated Solution Plan

The execution plan is dynamically shown, while constructed by the engine, in the *Execution Plan* window.

This window has a left panel for execution management and a right panel where the execution plan is shown (see figure 9.10).

Asking Information About Data Values

When an operator is selected, get input's and output's current values by clicking the *Info* button. A dialog window pops up (see figure 9.57) with a list of operator's typed input and output. Select one of them then click the *Get* button to receive value information from the engine (see figure 9.58).

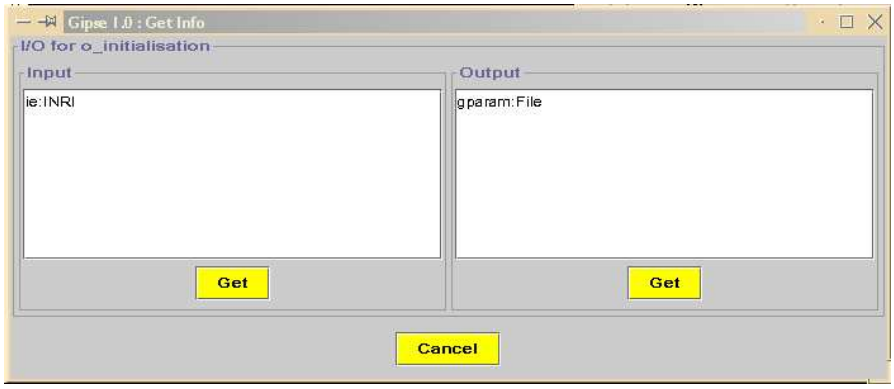


Figure 9.57: Asking information on data values

Figure 9.58: Getting information on data values

Backtracking

During execution the engine may backtrack when a problem (i.e. bad result evaluation) has been detected.

Each backtrack yields a new level in the dynamically constructed execution plan. Same level operators are all of the same color.

Three colors are provided to differentiate levels : by default green, pink and cyan.

Since an operator can be used several times at execution, it gets a unique state number each time it is used. The state number is drawn below the primitive operator in the execution plan. The list of successive state numbers is drawn below the operators in the Execution Tree (see figure 9.59).

9.5.3 Modification of the Knowledge Base During Execution

During execution, the operators currently processed by the engine are highlighted. When used once they are colored in a first level flag color. Then when used again, they turn a

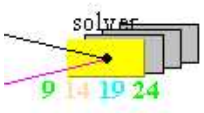


Figure 9.59: Operator's state numbers when backtracking

second level flag color. This helps identify operators that have been used a lot by the engine during execution.

9.5.4 The Dynamic Execution Tree

Highlights and flag colors are used the same way in the Execution Tree.

Each state number has the color of the corresponding backtrack level.

Moreover, an operator that has been used a number of times gets the same number of colored cards lying behind it (see figure 9.59).

9.5.5 Execution Logs

A log window is popped up at debug level greater than 0 (see figure 9.11). It is divided in three panels *General Log*, *Execution Log* and *Rule Log*.

The first one accumulates all logs coming from the engine. The second one shows only execution orders while the third one displays applied rules.

The last two have a *Save* button that allows the expert to save their contents in an ascii file.

9.5.6 Communication Interface Engine

At the end of the execution the engine asks the expert for the execution of a new request. In the case of a negative answer, a confirming message appears.

9.6 Customize your Interface

GIPSE allows the expert to change for her/his convenience a number of graphic attributes such as colors, shapes, size of operators.

Upon completion of changes, the expert may save them in a configuration file that can be reapplied.

Use the *View* command in the *Show* menu or the *View* button on the right vertical buttons bar (see previous figure 9.14) to access display customization.

9.6.1 Knowledge Base Display Customization

The *Knowledge Base* tab must be selected to customize the knowledge base.

After using the *View* command a dialog window pops up. This window has four tabs, each customizing a few features.

Primitive, Composite Operators

Use the first two tabs (see figure 9.61) to customize the look of an operator. Change its size with two sliders, its color (use the *Color* combo box or the *New Color* button) and its shape (use the *Shape* combo box).

Clicking the *New Color* button pops up a dialog window that allows creation of color by using three sliders (red, green, blue) (see figure 9.60). Simply click *OK* upon completion to go back to the view window.

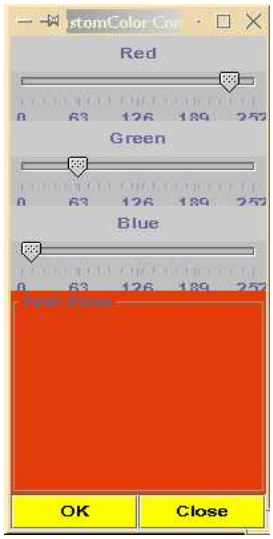


Figure 9.60: Setting a new color

After completing changes, click the *Apply* button. Changes are immediately taken into account.

Links

Use the *Link* tab (see figure 9.62) to change the look of the links. First choose the type of link using the *Type* radio buttons panel. Change the color of either regular links (sequential), choice or optional. Then use the *Color* combo box to choose the color.

Figure 9.61: Primitive tab

After completing changes, click the *Apply* button. Changes are immediately taken into account.

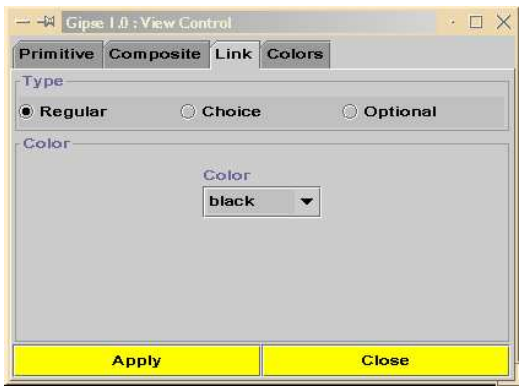


Figure 9.62: Link tab

Knowledge Base Colors

Use the *Colors* tab (see figure 9.63) to change colors. Change the color of selection for primitive operators, composite operators and links using the *Selection Colors* combo boxes.

Using the *Node Info Colors* combo boxes, change the color of the operator's labels, the input-output written next to an operator, the rule string written below and the data flow arrow.

After completing changes, click the *Apply* button. Changes are immediately taken into account.

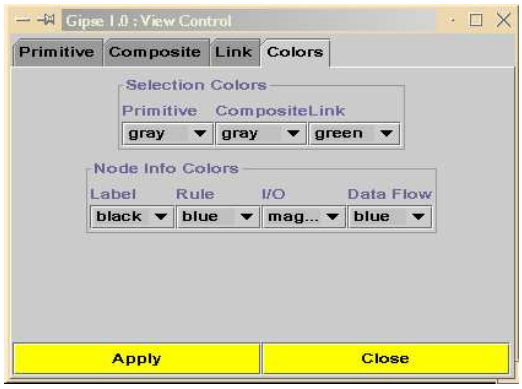


Figure 9.63: Colors tab

9.6.2 Execution Display Customization

Select the *Execution Tree* tab to customize the execution display.

After using the *View* command a dialog window pops up (see figure 9.64).

Figure 9.64: Execution view dialog

Only colors are customizable for the execution process.

The dialog window has three combo boxes panels for modifying highlight colors, flag colors and backtrack colors.

When the execution is running, the operator currently used by the engine (for expansion or execution) is highlighted in the knowledge base's graph and in the Execution Tree. Change this highlighting color for primitive or/and composite operators by using the appropiate combo boxes.

An operator used once by the engine turns a first level flag color ; then when used again, a second level flag color. Both in the knowledge base's graph and in the Execution Tree. Change these colors by using the appropiate combo boxes.

The engine may backtrack when looking for a solution. Each backtrack yields a new level in the dynamically constructed execution plan.

Three colors are provided (first level, second level, third level colors) that are successively used to color the operators of differents levels. Use the appropriate combo boxes to change these colors.

In the Execution Tree, an operator visited several times by the engine has cards drawn behind it. The card color can also be changed using the appropiate combo box.

To validate these changes, click the *Apply* button.

9.6.3 Saving a Display Configuration

After completing changes, save them in a configuration file. Use the *Save As* or *Save* command in the *Conf* menu to do so.

Use the *Open* command to reopen this configuration file.

Note that changed spacing between operators (*Increase X Spacing*, *Increase Y Spacing* commands) is also saved in the configuration file.

9.6.4 Opening a Display Configuration

To display a knowledge base according to a configuration file when loading, open the configuration file using the *Open* command then open the knowledge base.

When a base is already opened, apply a configuration file by loading it and using the *Repaint* command.

9.6.5 Coming Back to the System Default Display

Use the *Init* then *Repaint* commands to go back to the GIPSE default display.

Chapter 10

Practice

10.1 Installation

Copy the installation CD-Rom contents in a new (empty) directory and change to this directory.

Now in your current directory you get an `install.sh` script, a `confverif.sh` script, a `Makefile` and a tar compressed file for PEGASE+ installation `pegase.tar.Z`.

First, chose an installation directory (say `/usr/local/Pegase`), where you have the right to write (or have the job done by your favorite superuser) and where all the users of PEGASE+ can read from.

Then run `install.sh` with your installation directory as argument:

```
> install.sh /usr/local/Pegase
```

The scripts execute and first check your configuration, i.e. the version of your C++ compiler, of the librairies, of Java (if you get the GUI), of the make utility, etc. If the configuration is not correct the installation stops. Otherwise the scripts go on. They may create the installation directory if it does not exist (and if it is possible). Finally, the `pegase.tar.Z` is automatically uncompressed in the installation directory and you'll obtain the following organisation:

One file:

- a `yakl.el` file for the emacs yakl mode

Several directories:

- `lib` directory, which contains libraries:

  - the PEGASE+ engine library (`libpegase.a` and/or `libpegase.so`)
  - the support library (`libBlockssupport.a` and/or `.so`)
  - the `.jar` file for the graphic interface (optional).

- `bin` directory, which contains binary files:
  - the executable `parser` (static and/or dynamic version), to parse `.yakl` files and to generate C++ files.
- `include` directory, which contains all the necessary `.h` files in order to link a KB and the engine library (plus some `.cc` files necessary for template classes).
- `Doc` directory (optional), which contains this documentation.
- some usefull scripts (`.sh`).
- `Example` directory, which contains a KB example and utilities
  - a `Makefile`
  - the `launch.sh` script to launch a KB execution with the GUI.
  - some `.yakl` example files.

  The `Makefile` and `sort-yakl.sh` script should be copied each time you want to create your own KB (and `launch.sh` if you use the GUI). Don't remove or modify them!

You can now browse the the `Example` directory and try `(g)make` in it

```
> cd /usr/local/Pegase/Example
> make
```

And finally execute the resulting executable KBS (XXX) to see the example run.

```
> XXX
```

10.2 New KBS Creation

Create one of your own directory and name it about the name of the KB application you want to develop. In this directory, copy the `Makefile` and `sort-yakl.sh` script (and `launch.sh` script if you use the GUI) from the `Example` directory.

Create your own `.kb` and `.yakl` files, following the indications of chapter 2, and parse them (see below).

10.3 Use of the Parser

10.3.1 Emacs Mode

The emacs mode provides, by means of help menus, some facilities to write Yakl files. It can be obtained by modifying your `.emacs` file. Either you already have in your `.emacs` file a modification of auto-mode-alist variable, e.g.:

```
(setq auto-mode-alist
(append  '(
        ; Lisp and Lisp dialects modes
        ("\\.ll$"       .       lisp-mode)
        ; TeX and LaTeX related modes
        ("\\.tex$"      .       extended-LaTeX-mode)
) auto-mode-alist
))
```

In this case you just insert the 2 lines for yakl:

```
(setq auto-mode-alist
(append  '(
        ; Lisp and Lisp dialects modes
        ("\\.ll$"       .       lisp-mode)
        ; TeX and LaTeX related modes
        ("\\.tex$"      .       extended-LaTeX-mode)
        ; Yakl mode
        ("\\.yakl"       .      yakl-mode)
        ("\\.kb"       .      yakl-mode)
) auto-mode-alist
))
```

Or you add the modification of auto-mode-alist:

```
(setq auto-mode-alist
(append  '(
        ; Yakl mode
        ("\\.yakl"       .      yakl-mode)
        ("\\.kb"       .      yakl-mode)
) auto-mode-alist
))
```

And also:

```
 (autoload 'yakl-mode
      "/usr/common/Yakl/Emacs/yakl.el"
      "Yet Another Knowledge base Language" nil nil)
```

where `yakl.el` is the yakl mode file provided in the package (with the parser, etc.) that has to be stored in a common directory *e.g.*, here in /usr/common/Yakl/Emacs/yakl.el. This has to be done once.

Afterwards, each time you call emacs on a `.yakl` file, a new menu, named "Language", appears in your menu bar. Calling its sub-menus will ask you questions to fill syntax patterns for the different parts of a YAKL description of a knowledge base.

10.3.2 Parsing a file

In your KB directory, the command: parser `file.yakl` if it succeeds, generates several files: `file.save, file.h, file.cc, mainfile.cc, mainfile_aux.cc`.

If the parsing does not success, you may get two types of warning messages(***Warning and *Warning) three stars warnings are more "dangerous" than one star ones, i.e. they will more probably lead to errors that prevent from executing the KBS. The -w option to the parser command inhibits "one star" warnings.

If you use emacs, the error messages during parsing are printed in your compilation buffer and a click on an error line will display the error line in its `.yakl` file.

At the beginning it could be more convenient to parse files individually. Afterwards, the `Makefile` will do the job. Note that the automatic parsing implemented in the `Makefile` parses **all** the `.yakl` files that are presnet in your KB directory, beeing they indicated in the `.kb` file or not.

10.4 KBS Use

Once all files have been parsed execute `make` in your KB directory.

The command `make` triggers the `Makefile` and automatically launches when necessary (i.e. when files have been modified) parsing, compilation, etc. The generated executable KBS is named XXX by default, you can rename it by editiong and modifying the corresponding line in the `Makefile`.

Some options of PEGASE+ are known by the generated executable KBS:

- -d x (x= debug level)
- -r x (x= request number)
- -t (trace)
- -s (simulation mode)
- -g (GUI interface on)

10.4.1 Debug modes

Before the execution of a KBS the user is asked to choose a debug level. Each level provides different kinds of informations about what is happening, in the form of traces or messages. Below are detailed the possible levels.

- Level 0 provides no information, except fatal errors that stop the reasoning.
- Level 1 displays the execution plan, the beginning and end of execution or decomposition of operators, the calling syntax of primitive operators, the names of the executed rules, the results of the execution of pre/post conditions and effects. It also traces the execution of initialisation, assessment, repair and adjustment criteria.
- In addition, level 2 performs a step by step execution, and gives more details about rules that have been rejected, steps of repair mechanism and applicability of optional operators.
- Level 3 is similar to level 2, but the text of the applied rules is displayed in the GUI.

Contents

I Reference Manual **5**

1 The Program Supervision Problem **7**
1.1 Motivations . . . 7
1.2 Analysis of the Program Supervision Activity . . . 8
1.2.1 Users of a Program Supervision System . . . 9
1.2.2 Our Approach to Program Supervision . . . 10
1.2.3 The LAMA Platform . . . 11
1.2.4 Formal Definition . . . 12
1.3 Knowledge-Based Techniques for Program Supervision . . . 13
1.4 Knowledge-Based Program Supervision System . . . 14

2 Program Supervision Model **17**
2.1 Proposed Knowledge Model in Program Supervision . . . 17
2.1.1 Model Ontology Summary . . . 17
2.1.2 Supervision Operators . . . 19
2.1.3 Arguments . . . 21
2.1.4 Criteria . . . 21
2.1.5 Data and Domain Objects . . . 23
2.1.6 Functionalities and Requests . . . 23
2.1.7 Interrelations of Concepts . . . 24
2.2 Model of Problem-Solving Mechanism . . . 25
2.2.1 Formalisation . . . 28

3 Introduction to the YAKL Language **29**
3.1 Elements of the Language . . . 29
3.1.1 Operators and Arguments . . . 30
3.1.2 Criteria . . . 32
3.2 Use of the Language . . . 33

4 Inside PEGASE+ 35
4.1 Components of a PEGASE+ KBS . . . 35
4.1.1 Engine . . . 35
4.1.2 Knowledge Base . . . 36
4.1.3 The Fact Base . . . 38
4.1.4 Connection with real Programs . . . 39
4.2 The PEGASE+ PS Engine . . . 39
4.2.1 Main Loop . . . 39
4.2.2 Identification Phase . . . 40
4.2.3 Construction Phase . . . 42
4.2.4 Execution Phase . . . 43
4.2.5 Evaluation Phase . . . 44
4.2.6 Repair Phase . . . 45
4.2.7 Error Messages . . . 47

II User's Manual 49

5 Yakl Grammar 51
5.1 Start . . . 52
5.2 Global KB Description . . . 53
5.3 Regular Files . . . 54
5.3.1 Imported files . . . 54
5.3.2 External Declarations . . . 54
5.3.3 Definition of Argument Types and Instances . . . 55
5.3.4 Definition of Domain Types and Objects . . . 60
5.3.5 Definition of Functionalities and Requests . . . 62
5.3.6 Definition of Supervision Operators . . . 64

6 Methodology of Knowledge Base Development with PEGASE+ and YAKL 79
6.1 First, Define Sub-Problems . . . 79
6.2 Second, Add Strategic Criteria . . . 81
6.3 Complete and Refine the Base . . . 81
6.4 Guidelines . . . 82
6.5 Validation Facilities . . . 84
6.6 Requirements for Using Program Supervision Techniques . . . 84
6.6.1 Program properties . . . 84
6.6.2 Argument properties . . . 85
6.6.3 Composite operator properties . . . 85
6.6.4 Criteria properties . . . 85
6.7 Summary . . . 85

7 Example of Knowledge Base Development 87
7.1 Starting a new Knowledge Base . . . 87
7.1.1 Defining a Primitive Supervision Operator . . . 88
7.1.2 Defining a Composite Supervision Operator . . . 91
7.1.3 Defining Types . . . 94
7.1.4 Add Strategic Criteria . . . 98
7.1.5 Choice Criteria . . . 98
7.1.6 Initialisation Criteria . . . 100
7.2 Complete the Base . . . 101
7.2.1 Import clauses . . . 101
7.2.2 Defining a Functionality and a Request . . . 101
7.2.3 Defining a Knowledge Base . . . 103
7.3 Refine the Base . . . 104
7.3.1 Optional Operators in Sequences . . . 104
7.3.2 More on I-O Relations . . . 105
7.3.3 More on Alternative Decompositions . . . 106
7.3.4 Effects . . . 106
7.3.5 Attribute Value from a File . . . 107
7.3.6 Assessment Criteria . . . 108
7.3.7 Repair Strategy . . . 109
7.3.8 More on Assessment . . . 114
7.3.9 Parameter Flow . . . 114
7.3.10 Methods for Expert Types . . . 115

8 Detailed Example of a Simple Knowledge Base 117
8.1 KB definition . . . 117
8.2 Type and Domain Object Definitions . . . 118
8.3 Operators . . . 119
8.4 Request . . . 123

9 Graphic Interface 125
9.1 General Overview of GIPSE . . . 125
9.1.1 On Line Help . . . 125
9.1.2 Opening a Knowledge Base . . . 128
9.1.3 Visualisation of a Knowledge Base : tree vs. graph . . . 129
9.1.4 Executing a Request . . . 130
9.1.5 Exiting GIPSE . . . 133
9.2 Visualisation Tools . . . 133
9.2.1 Manipulating the Knowledge Base Graph . . . 133
9.2.2 Getting Information on Operators . . . 135
9.2.3 Data Types . . . 140
9.2.4 Displaying Types . . . 140
9.2.5 Loading a Type . . . 141

9.3 Creating and Modifying a Knowledge Base . . . 141
9.3.1 The New-Modify Bar . . . 141
9.3.2 Saving a Knowledge Base . . . 146
9.3.3 Creating or Modifying a Knowledge Base Type and Saving It . . . 146
9.4 Adding text and arrow on your Knowledge Base . . . 148
9.4.1 The Graphic Features Bar . . . 148
9.4.2 Moving Graphic Features Around . . . 149
9.5 Execution . . . 150
9.5.1 Selection of a Request . . . 150
9.5.2 Generated Solution Plan . . . 150
9.5.3 Modification of the Knowledge Base During Execution . . . 151
9.5.4 The Dynamic Execution Tree . . . 152
9.5.5 Execution Logs . . . 152
9.5.6 Communication Interface Engine . . . 152
9.6 Customize your Interface . . . 152
9.6.1 Knowledge Base Display Customization . . . 153
9.6.2 Execution Display Customization . . . 155
9.6.3 Saving a Display Configuration . . . 156
9.6.4 Opening a Display Configuration . . . 156
9.6.5 Coming Back to the System Default Display . . . 156

10 Practice **157**
10.1 Installation . . . 157
10.2 New KBS Creation . . . 158
10.3 Use of the Parser . . . 159
10.3.1 Emacs Mode . . . 159
10.3.2 Parsing a file . . . 160
10.4 KBS Use . . . 160
10.4.1 Debug modes . . . 161

A YAKL Semantics **171**
A.1 Syntactic Domains . . . 171
A.2 Semantic Domains . . . 172
A.3 Semantic Function . . . 173
A.3.1 Alternative and Sequence . . . 173
A.3.2 Details on Repair . . . 174

B YAKL Parser Error Messages **175**
B.1 YAKL Verification . . . 175
B.1.1 General Verifications . . . 175
B.1.2 Verifications on Operators, Functionalities, etc. . . . 175
B.1.3 Verifications on Rules . . . 176
B.2 Error and Warning Messages . . . 177

B.2.1 Error Messages 178
B.2.2 Three Star Warnings 185
B.2.3 One Star Warnings 186

Index

Achieved by, 63
Adjustment criteria, 78
Assessment criteria, 76
Body, 69
Call, 67
Composite Operator, 69
Functionality, 63
Initialization criteria, 75
Input Data, 63
Object Type, 61
Output Data, 63
Primitive Operator, 67
Repair criteria, 77
Request, 63
Achieved by, 102
Adjustment criteria, 110
Argument Type, 95
Argument Instance, 59
Argument Type, 55
Assessment criteria, 108, 114
Attributes, 56
Authors, 53
Body, 92
Call, 90
Characteristics, 64, 88
Choice criteria, 99
Choice criteria, 68
Code Files, 53
Composite Operator, 91
Distribution, 68, 92
Effects, 64, 106, 107
Extern, 55
Flow, 68, 92
Functionality, 102
I-O Relations, 105
I-O relations, 62, 89
Import, 54, 101
Initialisation criteria, 100
Input Data, 88
Input Parameters, 88
KB Path, 53
List of Files, 53
Methods, 56, 115
Object Instance, 61, 97
Object Type, 96
Optional criteria for, 68, 105
Output Data, 88
Override, 56
Parameter Flow, 114
Postconditions, 64
Preconditions, 64, 89
Repair criteria, 110
Root Node, 53
Set of, 57
Subtype Of, 55
calculation, 58
item, 58, 65, 107
re_execute, 109
send_down, 111
send_operator, 111
send_up, 111, 113
valid, 65, 71

action
    rule, 98
adjustment criteria, 77
adjustment method, 77, 110

adjustment rule, 109, 110
adjustment step, 78
alternative decomposition, 91, 93, 106
argument, 18, 21
argument instance, 96
argument type, 55, 94, 96, 118
assessment conditions, 71
assessment criteria, 75, 114
assessment rule, 108
assessment rule actions, 75
author, 88

back_choice, 46
backtracking, 109, 111
BNF, 171
BNF(Bacchus Naur Form), 51
body, 91

calculation, 101
characteristics, 19, 88
choice criteria, 73, 98
choice rules, 73
code file, 104
code in rules, 72
collection, 57
composite operator, 19, 68, 91, 120
conclusion
    rule, 33
construction phase
    engine, 25, 42
criteria, 19, 21, 28, 98
    adjustment, 22
    assessment, 22
    choice, 22
    initialisation , 22
    optionality, 22
    repair, 22

daemon, 37, 56, 58
data, 18, 21, 89
data flow, 91, 93
data instance, 38
declarations
    rule, 33
default value, 55
default value, 57
domain object, 18, 23, 38, 60, 118
domain object instance, 97
domain type, 60, 96

effects, 19, 106
end-user, 9
engine, 14
    main loop, 39
    Pegase+, 35, 39
    rule, 37
engine algorithm
    Pegase+, 39
evaluation phase
    engine, 25, 44
execution, 66
execution phase
    engine, 25, 43
expert, 8, 9, 87

facet, 37
fact base, 23, 38
frame, 36
functionality, 18, 19, 23, 62, 101

identification phase
    engine, 25, 40
import clause, 101
initialisation criteria, 74, 100
initialisation rule actions, 74

judgement, 71, 75

KB path, 103
knowledge base, 15, 36, 53, 103, 117
knowledge-based system, 10

Lama, 11
line
    item, 66

main loop

engine, 39
Matlab, 66
method, 94, 103, 115
adjustment, 46

operator, 18, 19, 64
optional criteria, 73
optional operator, 104
optional rule actions, 73
optionality criteria, 104
output data, 89

parallel decomposition, 91
parameter, 21, 89
parameter flow, 69
Pegase+, 35
engine, 39
plan, 18, 39
planning, 39
postconditions, 19
precondition, 89
preconditions, 19
predefined type, 55, 89
premise
rule, 33, 98
Primitive Operator, 88
primitive operator, 19, 66, 88, 119
program supervision, 8

range, 46, 55, 58, 77, 89, 107, 110
re_execute, 46
real execution, 90
repair criteria, 76
repair phase
engine, 26, 45
repair rule, 109, 111
repair rule actions, 76
request, 18, 23, 63, 101, 123
root node, 53, 103
rule, 32, 70
engine, 33
rule actions, 72
rule base, 37, 98
rule premise, 71

send_down, 45
send_operator, 46
send_up, 45
sequence decomposition, 91, 92
simulation execution, 90
slot, 37
state tree, 38, 109
syntax, 90
call, 66, 90

validation, 84
verifications, 175

Appendix A

YAKL Semantics

The denotational semantics of YAKL is presented in this appendix (its syntax has been presented in chapter 5). The semantics has been formalized for PEGASE+ use. Indeed, it should be noted that the same syntax may correspond to different interpretations. In our case, the process that interprets the language is in fact the engine, thus one semantics –corresponding to one interpretation— corresponds to one particular engine.

This semantics should cover all aspects of program supervision, that is execution of operators and of their attached criteria.

A.1 Syntactic Domains

Denotational semantics gives a meaning to the execution of YAKL supervision operators. Here, we focus on the meaning of operator execution called computations. First, we define the language of such computations in BNF short form (* denotes a repetition, () optional parts, and <> a pair):

computation $<$ **operator**, $\mathcal{R}_{\textbf{opname}} >$		*a computation = an op. + its reparation)*
operator :	ident	*a primitive op. is referred to by its name*
	| || (**computation** (, **computation**)*	*alternative composite*
	| **computation** ; **computation**	*sequence composite*
opname :	ident	

The computation of an operator is basically the pair composed of the operator and its attached reparation, because ....

The set of computations is noted $\mathcal{C}$omputation.

A.2 Semantic Domains

Let Ω be the set of supervision operators available in a particualr application. Ω is composed of primitive and composite operators. Let $Vars$ be the set of variables (instances of classes) manipulated by operators and $Values$ the set of all possible values for the variables. We consider that operators are executed in an environment, composed of variables and their current and past values (it is necessary to store past valuations for backtracking during reparation). At a given instant the supervision knowledge-based system is running and one operator is being executed in the environment created by the previous operator executions (or by the intial data given by the user's request). We denote $\mathcal{E}$ the environment, simply viewed as a stack. It is labelled by the name of the current operator and it contains all the valuations of variables: $[\Omega \rightarrow \Sigma] \times \mathcal{E}$ with Σ the set of all valuation functions (Σ: $[Vars \rightarrow Values_{\perp}]$; we denote $\sigma \in \Sigma$ a valuation function: $Vars \rightarrow Values_{\perp}$.

In order to indicate the place (in the environment stack) to go after an execution step, we introduce $Cont$ a continuation $Cont : [\mathcal{E} \rightarrow \mathcal{E}]$; $\phi \in Cont$ denotes a particular continuation function, $\phi : \mathcal{E} \rightarrow \mathcal{E}$.

Each supervision operator is labeled ($\forall \omega \in \Omega, l_\omega$ is the corresponding label, i.e. the name of the operator) and has an attached (possibly empty) reparation (R_ω) which may be either a local adjustment $\mathcal{A}_\omega$ or a global problem transmission $\mathcal{T}_\omega$. Moreover, ω has an initialisation function denoted $\mathcal{I}_\omega$.

We define two usual functions to handle environments: *put* to add an element (label and valuation) to the environment and *get*, to get an element from the environment; in the sequel, ϕ stands for the current continuation and ρ for the current environment:
$put : \mathcal{E} \times \Omega \times \Sigma \rightarrow \mathcal{E}$
$put(\rho, \omega, \sigma) = \rho \times (l_\omega, \sigma)$, where $\times$ means addition on the environment stack of a new pair (label,valuation)

$get : \mathcal{E} \times \Omega \rightarrow \mathcal{E}$
$get(\rho, \omega) =$ if $\rho = \rho' \times (l_\omega, \sigma)$ then ρ else $get(\rho', \omega)$

For each operator ω, its attached common criteria are interpreted as follows:
$\mathcal{I}_\omega : \quad \Sigma \rightarrow \Sigma$
$\mathcal{A}_\omega : \quad \Sigma \rightarrow \Sigma$

We denote $|\omega|$ the result of the effective execution of the supervision operator (e.g., execution of the program for a primitive one).
$|\omega| : \quad \Sigma \rightarrow \Sigma$

For simplicity reasons we also define **E**, a function which combines the execution of an operator and the check whether the result has to be repaired (based on assessment criteria).
E: $\Omega \rightarrow \mathcal{E} \rightarrow$ Bool (true means the result is correct, false, that the result is not and the operator must be repaired).
In the sequel, E_ω will denote $\mathbf{E}(\omega)$.

A.3 Semantic Function

We define a semantic function $[\![]\!]$: $\mathcal{C}$omputation $\rightarrow Cont \rightarrow \mathcal{E} \rightarrow \mathcal{E} \cup \{error\}$
In addition, we define a complementary semantic function to interpret the repair criteria:
$\mathcal{R}[\![]\!]$: $\mathcal{C}$omputation $\rightarrow Cont \rightarrow \mathcal{E} \rightarrow \mathcal{E} \cup \{error\}$

The following holds for any type of operator (primitives or composite), here R_ω represents indifferently either $\mathcal{A}_\omega$ or $\mathcal{T}_\omega$. σ represents the current valuation at a given instant, in the current environment (the one at the "top" of the environment stack).

$[\![< \omega, R_\omega >]\!]\phi\rho =$
if $(E_\omega(put(\rho, \omega, \mathcal{I}_\omega(\sigma)) ==$ true) then $\phi\rho[(l_\omega, |\omega|(\sigma))/(l_\omega, \sigma)]$
else $\mathcal{R}[\![< \omega, R_\omega >]\!]\phi\rho$

Which means: if the result of ω is correct after its computation in the environment modified by the application of the initialisation criteria of ω (which leads to a new valuation), then use the continuation ϕ in an environment which is ρ where (l_ω, σ) has been replaced by $(l_\omega, |\omega|(\sigma))$ (the same label, but a different valuation resulting from the execution of ω). Else, in the case when reparation is necessary, interpret the repair criteria whith the complementary semantic function $\mathcal{R}[\![]\!]$ (see below).

This corresponds to the PEGASE+ problem-solving mechanism: first apply initialisation criteria, execute the operator (decompose it if composite), then assess the results, and repair if necessary, using the repair criteria.

A.3.1 Alternative and Sequence

Concerning composite operators, we denote $\omega_|$ an alternative decomposition and $\omega_;$ a sequential one.

Alternative operators have an attached choice criteria denoted by $\mathcal{C}_{\omega_|}$ which selects one sub-operator ($C_{\omega_|}$: $\Omega^* \rightarrow \Omega$):

$[\![< \omega_|(< \omega_1, R_1 >, ..., < \omega_n, R_n >), R_{\omega_|} >]\!]\phi\rho =$
$[\![< \omega_i, R_i >]\!]\{[\![< \omega_|, R_{\omega_|} > \phi\rho]\!]\}\rho$, where $\omega_i = \mathcal{C}_{\omega_|}(\omega_1, .., \omega_n)$.

The execution of an alternative operator is the execution of the selected sub operators followed by the execution of the reparation of the main operator. The equation tells that the meaning of an alternative operator is the meaning of its selected sub operators with the meaning of the repair criteria of the operator as continuation.

Sequences are treated as binary operations. An optionality partial function $\mathcal{O}$ is attached to each sequential operator. It corresponds to the optionality criteria of the operator; it does nothing for non optional sub-operators, otherwise, it decides, depending on the current environment, if the currently examined sub-operator in the sequence must be executed or not.

$\mathcal{O} : \Omega_{seq} \rightarrow \Omega \times \mathcal{E} \rightarrow \Omega \cup \{\text{NOP}\}$.
Ω_{seq} is the set of sequential operators, NOP stands for the cases when the optionality criteria decides to skip a sub-operator (not to execute it).
$\mathcal{O}(\omega_;)$ is denoted $\mathcal{O}_{\omega_;}$
$[\![< \omega_;(< \omega_1, R_1 >, < \omega_2, R_2 >), R_{\omega_;} >]\!]\phi\rho =$
$\quad [\![< \mathcal{O}_{\omega_;}(\omega_1), R_1 >]\!]\ \{[\![< \mathcal{O}_{\omega_;}(\omega_2), R_2 >]\!]\ \{[\![< \omega_;, R_{\omega_;} >]\!]\phi\rho\}\rho\}\rho$
$[\![<\text{NOP},R >]\!]\phi\rho = \phi\rho\ \forall\ R.$

A.3.2 Details on Repair

More specifically, the semantic equations of the complementary semantic function to interpret the repair criteria $\mathcal{R}[\![]\!]$ decompose into two cases, depending whether the reparation corresponds to an adjustment ($R_\omega = \mathcal{A}_\omega$) or to a global problem transmission ($R_\omega = \mathcal{T}_\omega$).

Adjustment
$\mathcal{R}[\![< \omega, \mathcal{A}_\omega >]\!]\phi\rho =$
if $(E_\omega(\rho)) ==$ true then $\phi\rho[(l_\omega, |\omega|(\sigma))/(l_\omega, \sigma)]$ else $[\![< \omega, \mathcal{A}_\omega >]\!]\phi\rho[(l_\omega, \mathcal{A}_\omega(\sigma))/(l_\omega, \sigma)]$

If, after execution in environment ρ, the operator needs not to be repaired, then use the continuation ϕ in an environment which is ρ where (l_ω, σ) has been replaced by $(l_\omega, |\omega|(\sigma))$ (the same label, but a different valuation resulting from the execution of ω with initial σ). Else re-execute the same operator, with a different environment as modified by the $\mathcal{A}_\omega$ function (ρ in which (l_ω, σ) has been replaced by $(l_\omega, \mathcal{A}_\omega(\sigma))$), which corresponds to applying adjustment criteria before re-executing an operator in PEGASE+.

Problem Transmission
$\mathcal{R}[\![< \omega, \mathcal{T}_{\omega<\omega_i, R_i>} >]\!]\phi\rho =$
if $get(\rho, \omega_i) = \emptyset$ then $error$ else $[\![< \omega_i, R_i >]\!]\phi\ get(\rho, \omega_i)$

If the operator to which the problem is to be transmitted (ω_i) does not exist in the current environment (i.e. it has not been executed in the past by the system), raise an error, else execute the ω_i operator, in the environment as it was just before the computation of ω_i.

Appendix B

YAKL Parser Error Messages

B.1 YAKL Verification

YAKL is intended to provide a rather strict framework that avoids most programming bugs and errors in the generated executable code of the knowledge base (which will be linked with the engine to constitute a program supervision KBS). During the parsing of a knowledge base, some verifications are done. First, the parser enforces the YAKL syntax, as described in this chapter, and second, some more semantical checks are performed. Here is a list of the syntactic and semantic verifications that are done during the parsing of a knowledge base, they produce either errors which stops the parsing or warning messages that may be fixed later. But, except for specific purposes, warning messages in a knowledge base imply that there will be an **execution error**!

B.1.1 General Verifications

- Verifications that (imported) files exist
- Type checking in assignments,
- Type compatibility between argument value type and default value or range,

B.1.2 Verifications on Operators, Functionalities, etc.

- Compatibility of arguments of functionalities (in number and type) with the arguments of the different operators that can be used to achieve them.

  - The number of input and output arguments of an operator must be greater or equal than the number of input and output arguments of its functionality.
  - and their types must be the same (or a subtype in operators)

- Check arguments of optional operators in a sequence, which must have the same number and type of input and output arguments.

- Data flows verifications:

  - Completeness verification of data flows *i.e.* all data of both a father operator and its sons should be involved in a data flow (distribution or flow), all father operator input are used by at least one sub-operator, all father operator output data receives a value from a sub-operator output, etc.
  - Mode (in/out) and type compatibility checking in distribution and flow.

B.1.3 Verifications on Rules

- Actions in the preconditions part are not allowed, and nor conditions in the actions part.

- Depending on the type of the rules (choice, evaluation, ...) the vocabulary allowed in its premise or actions are different (according to the syntax described in this chapter).

- Rules related to an operator may only refer to attributes of this operator or sub-operators, or to a (global) domain object. For instance, choice actions should only make reference to a sub-operator, evaluation rules should assess either the operator itself or one of its arguments, or one of its sub-operators;

- Any referenced object (data, domain object, operator, etc.) must exist, and any referenced attribute must exist and belong to the indicated object.

- The rules in a rule base that are intended for a specific purpose (choice, evaluation, initialisation or adjustment), must fulfil their purpose, *e.g.,*

  - Any parameter must have at least one initialisation rule when a default value has not been supplied; *Otherwise it may produce an execution error.*
  - Choice criteria have to activate all the different sub-operators of an alternative decomposition, under mutually exclusive conditions;
  - Adjustment criteria have to provide means of adjusting all parameters. (*Not yet checked by current parser*)
  - When repairing, the set of rules that propagate the problems must conform a complete path from the operator which generates the problem to the faulty operator. (*Not yet checked by current parser*)

- Some rule bases have complementary roles, for instance:
  - The set of of premises of adjustment rules must be consistent with the set of conclusions of evaluation rules (consistent naming of judgements). (*Not yet checked by current parser*)
  - The same thing must hold for reparation rules *i.e.* the naming of problems along this path must be consistent. (*Not yet checked by current parser*)

Other verifications may concern the completeness of the knowledge, for instance, operator hierarchies must have primitive operators as leaves.

B.2 Error and Warning Messages

Following the principles decribed in the previous sections, error or warning messages are displayed to users, in case they do not follow the expected syntax and semantics. These messages are listed in alphabetical order and explained in the following tables, using the same notation conventions as in the chapter about YAKL grammar (chapter 5).

B.2.1 Error Messages

$\mathcal{A}$

"Already defined argument in current operator/functionality" IDENT
Attempt to create a new argument, named IDENT attached to an operator (or a functionality), but IDENT has already been used to name another argument in the same operator/functionality, hence a name clash
"Already defined argument instance" IDENT
Attempt to create a new argument instance, named IDENT, but IDENT has already been used to name another argument instance, hence a name clash
"Already defined attribute in current type" IDENT
Attempt to create a new attribute, named IDENT attached to an expert type, but IDENT has already been used to name another attribute in the same type, hence a name clash
"Already defined attribute (no override for a set)" IDENT
Attempt to override an attribute of type collection (defined as Set of..)
"Already defined name" IDENT
Attempt to give a name (to an operator, a functionality, etc.) that has been already used for the same sort of element.
"Already defined rule in current operator" IDENT
Attempt to create a new rule, named IDENT attached to an operator, but IDENT has already been used to name another rule in the same operator, hence a name clash (it is possible to have the same names only for rules belonging to 2 different operators).
"Already defined object instance" IDENT
Attempt to create a new domain object instance, named IDENT, but IDENT has already been used to name another object instance, hence a name clash.
"Already defined type" IDENT
Attempt to define a new argument or object type the name of which has been already defined.
"Argument has not the expected mode" IDENT (expected input/output mode)
In pre (resp. post) conditions, involved arguments must be input (resp. output) ones. You cannot test output data before execution (in pre-conditions), because they do not have values yet, and normally input data do not change during execution, so testing them is useless in post conditions. The expected input or output mode is indicated into parentheses.

For "Already defined ..." messages, the solution is to check your previous definitions, to find the homonym, decide which element should use this name and chose another one for the other.

C

"Cannot access an element in non-collection" IDENT
In an assignmemt, the notation ID(X), where X is an integer, should refer to the X^{th} element of a collection, but identifier IDENT does not correspond to a collection.
"Cannot print a collection as a whole" IDENT
To transform the syntax of a primitive operator into a correct command line, everyelement in the syntax must be "printable" i.e. of simple type, that can be accepted by the Unix shell, for instance. If there is a composite identifier in the syntax with one of its sub-identifiers (IDENT) corresponding to a collection, and if no index is provided (using the notation IDENT(X), where X is an integer), the whole collection is refered, and there is no standard means to print a collection.
"Choice criteria not allowed if no choice in body"
Attempt to use a choice criteria in a non choice composite operator.
"Composite identifiers not allowed in distribution" COMIDENT
Attempt to use a full composite identifier in a distribution flow (only the name of an operator followed by the name of one of its arguments is allowed).
"Composite identifiers not allowed in flows" COMIDENT
The same holds for data flows among children.
"Current object has no attribute named" IDENT
In a composed identifier, if the first sub-identifier refers to an object, attempt to reference an attribute name (IDENT) that does not belong to the object.
"Current operator does not decompose into" IDENT
In a distribution flow only the sub-operators of the current one (the parent) are allowed on the left side of the slash.
"Current operator has no argument named" IDENT
Attempt to reference, in a flow, an argument name (IDENT) that does not belong to the current operator.
"Current request has no attribute named" IDENT
In a request assignement, attempt to reference an attribute name (IDENT) that does not belong to the current request.

D

"Data field not assigned" IDENT
In a request definition check if all input data are assigned a value, i.e. if all fields of the corresponding functionality are filled.
"Data not allowed in parameter flows" IDENT
Data (as IDENT) are not allowed in parameter flows, use data flows instead.
"Default out of range" value
The value given as default for an attribute is out the given range (only checked for simple types: string, integer, float), so be carefull for other types!

$\mathcal{E}$

"Error in code" CODE
Use of a piece of code (CODE) that is not syntactically correct (in C++). Note that parsing in the code parts is very simple and other problems may occur at run-time.

$\mathcal{F}$

"File not previously saved?: " IDENT
The file (named IDENT) to restore cannot be found, as if it has not been saved before (the corresponding .save file does not exist).
"Functionality not achieved by this operator" IDENT
When checking if a functionality is achieved by an operator, IDENT refers to an operator declared by the functionality as achieving it, but which has no corresponding **Functionality** declaration.

$\mathcal{I}$

"Illegal path" STRING
The file path (STRING) given in KB descrition does not correspond to a legal path on the current machine.
"Impossible reference from" IDENT
In a composite identifier of the form "id_1.id_2.id_3. ...", one ofthe first identifiers (IDENT) is of a "terminal" type (with no attributes) so it is impossible to have a dot after it. Composite identifiers are reserved for structured objects.
"Incompatible in/out argument types for optional" IDENT
An optional sub-operator (named IDENT) of the current one should allow to "shortcut" it, i.e., if it does not apply, that is its input and output arguments should be in the same order and of the same type, to let input be passed unchanged to output for next operators in the sequence.
"Incompatible modes (In/Out) in flow between " op1.arg1 and op2.arg2
Mode clash in a data or a parameter flow: attempt to match two arguments of different operators that do not have compatible modes (arg1 must be an output data and arg2 an input data).
"Incompatible type/mode for functionality and operator argument" IDENT
When an operator claims to achieve a functionality it should have the same arguments, in the same order, and with the same types as its functionality. IDENT is an argument of the operator that has not the same type or mode (in/out) as in the corresponding functionality.
"Incompatible type, expected type:" IDENT
In assignments the right value and the left variable must have the same type (named IDENT) (type here is known, thus it is expected as the type of the left variable).
"Incompatible type with overriden attribute in supertype:" IDENT
In case of type hierarchy, you cannot change the type (named IDENT) of an overriden attribute. Note: for the moment only the if_needed daemon can be modified in an overridden attribute.

"Incompatible types in flow between " op1.att1 *"and"* op2.att2
Type clash in a data flow: attempt to match two attributes of different operators (att1 of op1 and att2 of op2) that do not have the same type.
"Invalid name, could collapse with language key-words" IDENT
IDENT is a symbol corresponding to a key-word of the underlying programming language (C++) or to a YAKL key word; avoid using such identifiers.
"Invalid reference" IDENT
In a composite identifier of the form "id_1.id_2.id_3. ...", one of the id_i (IDENT) cannot be referenced (for the 1st this means: it is not an attribute of current operator, nor a declared variable name in a rule, nor a domain object; for the other ones this means: they are not attributes or method names of their previous identifier in the composite,...).
"Invalid reference: " IDENT1 " is not an attribute/method of type " IDENT2
In the left part of an IO relation, or of an effect, a composite identifier involves an invalid sub-identifier (IDENT1) which is not a correct access for the type of IDENT2.

$\mathcal{N}$

"No affectation in distribution: " expression
Attempt to assign an expression to an attribute of operator in a distribution flow (from parent to children).
"Non printable type in script" IDENT
To transform the syntax of a primitive operator into a correct command line, every element in the syntax must be "printable" i.e. of simple type, that can be accepted by the Unix shell, for instance. Here, IDENT type is not printable in this sense.
"Not an argument of current operator" IDENT
Attempt to use an identifier as the name of an argument of the current operator while it is not (e.g. in a pre/post condition, in a rule, in effects, etc.).
"Not a child of current operator" IDENT
In a data flow among operators only the sub-operators of the current operator can appear.
"Not a collection attribute" IDENT
IDENT does not corresponds to the name of a collection attribute of a type. Attempt to use an argument as if it was a collection (i.e., initialise it with a set of values or with an empty collection (nil),) while it was not declared as a collection (Set of..).
"Not an input data of the current operator" IDENT
In an IO relation the right part must be (part of) an input data of the current operator, IDENT is not.
"Not an optional sub-operator of current" IDENT
Attempt to use a name (IDENT)as the name of an optional sub-operator of current operator, but this name does not correspond to a declared optional sub-operator.
"Not an output data of the current operator" IDENT
The identifier (IDENT) should correspond to an output data of the current operator(e.g., in an I-O relation).
"Not a parameter" IDENT
Attempt to use (e.g., in a parameter flow) a name of a non-parameter argument.
"Not a parameter/in data of current operator" IDENT
Attempt to use in a rule a name as if it was the one of a parameter or of an input data of current operator, while it is not (e.g., assign it by "init_child_parameter" ...)
"Not a parameter of current operator" IDENT
Attempt to use (in a rule) a name (IDENT) as if it was the one of a parameter of current operator, while it is not.
"Not the operator of current criteria" IDENT
In a rule the name (IDENT) used after the use_optional_operator command is not the name of the operator indicated in the "Optional Criteria for..." header.
"Nothing in calling syntax!"
The calling syntax of an operator must not be empty.

$\mathcal{M}$

"Mismatched argument with achieved functionality" IDENT
When an operator claims to achieve a functionality it should have the same names for for arguments as its functionality. IDENT is an argument of the operator that does not exist in the corresponding functionality.

$\mathcal{O}$

"Operator is not the current one" IDENT
In a distribution flow the operator on the left part of the slash should be the current one (the parent); children are on the right side.
"Optional criteria useless if no optional in body"
Attempt to use a optional criteria in a composite operator with no optional sub-operator.
"Overriding an unknown attribute in supertype(s):" IDENT
In case of type hierarchy, you cannot declare as overriden an attribute that does not belong to (one of) the supertype(s) Note: for the moment only the if_needed daemon can be modified in an overridden attribute.
"Owner's type does not match current definition" IDENT
Attempt to assign an operator (named IDENT) as the owner of a rule while not in the definition of this operator.

$\mathcal{P}$

"Parameters not allowed in data flows " op1.att1 and op2.att2
Attempt to use a parameter in a data flow (use Parameter Flow, or init_child_parameter instead).

$\mathcal{R}$

"Restoration failed (check for incompatible formats?)"
It was impossible to restore an imported file, it may be due to changes in internal saving format. Try to parse your whole KB again.
"Root operator not defined, will lead to EXECUTION ERROR" IDENT
The root operator(named IDENT), as decared in the KB description has not been defined in any of the .yakl KB files, the engine will be unable to work on this incomplete KB!
"Rule parse error before" $
It is the standard rule error message, when the parser totally fails. It may be due to a forgotten comma (between premisses or actions), or to a misuse of names of operator arguments, etc.

$\mathcal{S}$

"Should return a String" IDENT
File names or searched elements in items are usually indicated by plain strings, but instead you can use: the value of some argument (thus this value (IDENT) must be of type String) or the return value of a method (thus the method (named IDENT) must return a String).

$\mathcal{T}$

"Type mismatch between value and attribute" IDENT
In an assignement the left and right part types must be compatible. In particular if the left part (IDENT) corresponds to a collection attribute, the right part must be a set of values and vice versa. In a request a missing value for an argument also leads to this error.
"Type of collection element should be printable" IDENT
To transform the syntax of a primitive operator into a correct command line, everyelement in the syntax must be "printable". The collection element refered to by IDENT has not a simple type that can be accepted by the Unix shell, for instance.

$\mathcal{U}$

"Undefined object" IDENT
Attempt to intialise a collection with an object name (IDENT) that has not been defined yet. If you want a string use “”.
"Undefined operator argument" IDENT
Attempt to reference, in a data flow, an argument name (IDENT) that does not belong to the corresponding (sub-)operator.
"Undefined operator" IDENT
In a data flow among operators an undefined operator name (IDENT) appears.
"Undefined functionality" IDENT
When checking if a functionality is achieved by an operator, IDENT refers to an undefined functionality.
"Undefined supertype" IDENT
Attempt to define a sub type (argument or object type) of a non existing super type (IDENT).
"Undefined type" IDENT
Attempt to assign an unknown type (IDENT) to a variable in a rule or to an argument of an operator, etc.
"Undefined functionality" IDENT
Attempt to instantiate a request from an undefined functionality
"Undefined operator" IDENT
Attempt to use in a rule an operator name (IDENT) which is unknown (neither the current one nor an already defined one) e.g. to assess an operator, to favor an operator (use_operator, ...).
"Undefined operator or functionality" IDENT
Attempt to add an undefined operator (or functionality) (named IDENT) as a child of a composite operator.
"Unknown file" IDENT
In checking the existence of files declared in a KB, the name IDENT does not correspond to an existing file.

The diagnosis for most "Undefined ..." messages may be that the import clause of the file containing the needed definition is missing, check that point.

B.2.2 Three Star Warnings

Three stars warnings are more "dangerous" than one star ones, i.e. they will more probably lead to errors that prevent from executing the KBS, but they can be accepted in an intermediate state of the KB development.

*"*** Warning: Undefined functionality"* IDENT
Attempt to create a functionality instance (a request) from an undefined functionality (IDENT) (supposed to be not yet defined, but should be soon!).
*"*** Warning: No data flow/distribution for argument "* IDENT " of operator " IDENT
An argument (named IDENT) of a (sub)operator does not appear in a data flow or a distribution flow, so this means its value is not used outside the operator. Is it on purpose?
*"*** Warning: Parameter "* IDENT *" has no default nor initialisation rule."*
A parameter of the current operator has no means to initialise its value. You should write some initialisation rule or default value otherwise at execution there will be a problem.
*"*** Warning: repair without assessment, may never be active."*
A repair criteria has been defined but no assessment one, hence the assessment process will never occur and the repair rules will never be triggered; remove the repair criteria or add an assessment one.
*"*** Warning: Undefined operator/functionality"* IDENT
Attempt to send an unknown problem (named IDENT) in a rule to an operator (or functionality) (supposed to be not yet defined, but should be soon!)
*"*** Warning: Modification of input data:"* IDENT
In a rule, assignement of an input data: input data are given –by definition– and are not supposed to be modified. Shouldn't IDENT be a parameter?

B.2.3 One Star Warnings

They are real warning, in the sense that their presence will not prevent the KBS execution, but they may correspond to an oversight.

" Warning: argument(s) not in calling syntax"* list of IDENTs
In the calling syntax of an operator, the listed arguments of this operator do not appear, which may be correct but may also be a mistake.
" Warning: avoid parameters in data flows "* IDENT
Parameters (like IDENT) should appear in Parameter flow, or be used by init_child_parameter in rules.
" Warning: Don't forget to define method: "* IDENT1::IDENT2 ()... and to include file " file.h
When the expert declares a method (IDENT2)attached to a type (IDENT1) in the KB, the body of this method must be written somewhere (in a .cc file for instance), this is a reminder for the expert. Moreover, including the current file interface (file.h) is necessary to have access to the type definition when the KB will be compiled.
" Warning: IO relations ignored in composite operator for"* IDENT
Output arguments (like IDENT) of composite operator do not have IO relations (they are obtained from lower levels, where primitive operators do have some). They will be ignored.
" Warning: missing IO relations in terminal operator for"* IDENT
Output arguments (like IDENT) of a terminal operator are supposed to have IO relations, to indicate how they "derive" from input ones.
" Warning: uneffective left part of effect. Not an output data of the current operator:"* IDENT
Effects only apply on output data, since IDENT does not correspond to an output data of the of the current operator, these effects will be ignored for future processing.

Bibliography

[CAM+97] M. Crubézy, F. Aubry, S. Moisan, V. Chameroy, M. Thonnat, and R. di Paola. Managing Complex Processing of Medical Image Sequences by Program Supervision Techniques. In *SPIE International Symposium on Medical Imaging*, Newport Beach, California USA., February 1997.

[CMM98] M. Crubezy, M. Marcos, and S. Moisan. Experiments in Building Program Supervision Engines from Reusable Components. In *3th European Conference on Artificial Intelligence-ECAI98. Workshop on Applications of Ontologies and Problem-Solving Methods.*, August 1998.

[MMMV96] S. Mathieu-Marni, S. Moisan, and R. Vincent. A Knowledge-Based System for the Determination of Land Cover Mixing and the Classification of Multi-Spectral Satellite Imagery. *International Journal of Remote Sensing*, 17(8):1483–1492, May 1996.

[Moi98] S. Moisan. *Une plate-forme pour une programmation par composants de systèmes à base de connaissances.* Habilitation à diriger les recherches, Université de Nice, April 1998.

[SMV+99] C. Shekhar, S. Moisan, R. Vincent, P. Burlina, and R. Chellappa. Knowledge-based control of vision systems. *Image and Vision Computing*, 17(8):667–683, May 1999.

[TMC99] M. Thonnat, S. Moisan, and M. Crubézy. Experience in Integrating Image Processing Programs. In *Internatianal Conference on Computer Vision Systems*, Las Palmas, Canary Islands, January 1999.

[vdE96] J. van den Elst. *Modélisation de Connaissances pour le Pilotage de Programmes de Traitement d'Images.* Thèse, université de Nice-Sophia Antipolis, octobre 1996.

[vdEvHT95] J. van den Elst, F. van Harmelen, and M. Thonnat. Modelling Software Components for Reuse. In *Seventh International Conference on Software Engineering and Knowledge Engineering*, pages 350–357. Knowledge Systems Institute, June 1995.

Unité de recherche INRIA Sophia Antipolis
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399