What can Program Supervision do for Program Re-use?

Monique Thonnat, Sabine Moisan

What can Program Supervision do for Program Re-use?

M. Thonnat and S. Moisan

I.N.R.I.A. – B.P. 93
F-06902 Sophia Antipolis Cedex, France

Abstract. We are interested in knowledge-based techniques (called *program supervision*) for managing the re-use of a modular set of programs. The focus of this paper is to analyze which re-use problems program supervision techniques can solve. First we propose a general definition for program supervision, a knowledge representation model, and a reasoning model. Then we analyze program supervision solution for re-use in terms of the structure of the programs to re-use and in terms of the effort for building a program supervision knowledge base. The paper concludes with what program supervision can do for program re-use from the points of view of the code developers, the experts, and the end-users.

Keywords: Program supervision, software re-use, knowledge-based system

1 Introduction

We are interested in knowledge-based techniques (called *program supervision*) for managing the re-use of a modular set of programs. The role of program supervision is to select programs in an existing library, to run the programs for particular input data and eventually to control the quality of their results. Various knowledge-based systems have been developed for this purpose, notably in the domains of image processing [3], [2], signal processing [9] and automatic control [6], [7]. For a more detailed review see [13] which is a general review, [15] for software re-use in software engineering and [10] for software re-use in signal processing and automatic control.

The focus of this paper is to analyze which re-use problems program supervision techniques can solve. This analysis is presented in terms of the structure of the programs to re-use and in terms of the effort for building a program supervision knowledge base. After this introduction, section 2 explains our analysis of the re-use difficulties and proposes a general definition of program supervision. Section 3 then presents a knowledge representation model for program supervision, and introduces the major notions of primitive and complex operators and specialized criteria, while section 4 presents how this knowledge is used during the reasoning of a program supervision system. Section 5 details the conditions of use of program supervision techniques in the framework of this model. This paper concludes with what program supervision can do for program re-use from the points of view of the code developers, the experts, and the end-users.

2 Re-using a Set of Programs

The use of existing libraries of programs has become a critical resource in many disciplines. Numerous programs have been developed in domains like signal or image processing and scientific computing. The programs have been written by specialists in a particular domain and are intended to be applied by non-specialists in this domain. New programs implement more and more complex

functionalities and their use is more and more subtle. One drawback is that the non-specialist user must know how to choose programs depending on different purposes, how to run each program, and how to chain programs in the correct order to obtain a result. If it is too demanding for an end-user to catch the complexity of new programs, they will never be widely applied. When analyzing the activity of re-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. An end-user must make important efforts in order to have the data correctly processed, and to efficiently use the set of programs.

For an end-user faced to 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. to *build a model* of them. Afterwards, since a single program is not usually sufficient to solve a complex processing request, the end-user must figure out which programs can be combined together and how. That means to know how to choose which program comes 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 w.r.t. the data at hand. Then, to execute a chosen combination, the user has to actually run the programs, which implies to know 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 of 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 as good as expected, the user must infer which previously executed program is faulty, if it can be re-run with new parameter values and how to compute the values, or if it must be replaced by another program and in this case by which one.

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

Every end-user could not have such a deep understanding of the program semantics and syntaxes. One possible solution to this problem is to use a tool that transparently manages the processing complexity, in order to automate the easy re-use of the programs. Among different techniques for re-use, we propose *program supervision* techniques which aim at capturing the knowledge of program use and to free the user from the processing details. The objective of program supervision 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, the reception of a user's request as input produces as output the executions of the appropriate programs with their resulting data as shown in figure 1.

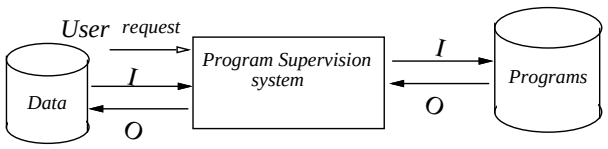

Fig. 1. A program supervision system helps a user to re-use a set of programs for solving a request on input data I to obtain output data O

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 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 (where the data flow is correct and the same program may appear several times)
- $\mathcal{O}$ a set of actual output data such that:
  - $\mathcal{O}$= $\Pi(\mathcal{I})$ and
  - $\mathcal{C}(\mathcal{O})$ holds.

We propose to emulate the strategy of an expert in the use of the programs by a knowledge-based system. A program supervision knowledge-based system, according to this formal definition, helps a non-specialist user apply the programs in different situations as shown in figure 2. It is composed of a program supervision engine and a knowledge base. The role of the program supervision engine is to use this knowledge for effective *planning*, *execution* and *control* of execution of the programs. The knowledge base contains the representations rp_i and rc_j of programs p_i, combinations of programs c_j, and a set of decision criteria cr_k. The contents of the representations rp_i, rc_j and of the criteria cr_k should be sufficient for the engine to select the programs, to initialize their parameters, to manage non trivial data-flow, and to combine the programs to produce a satisfactory plan Π depending on the input data, constraints, and request.

3 Program Supervision Knowledge Base Model

In this section we briefly present the main characteristics of a program supervision knowledge base.

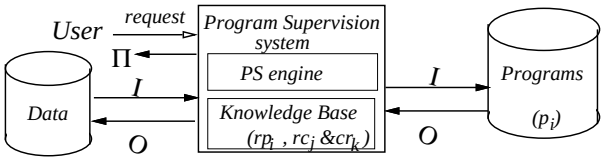

Fig. 2. A knowledge-based program supervision system helps a user to re-use a set of programs for solving a request on input data $\mathcal{I}$ to obtain output data $\mathcal{O}$, as the results of the execution of a plan Π. It is composed of a program supervision engine and a knowledge base. The knowledge 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.

As program supervision is a general problem arising in various application domains we are interested in providing both knowledge models and software tools which are independent of any particular application and of any library of programs. A program supervision model defines ways of describing programs for re-use, i.e. what structure reusable program descriptions must have 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.

Different categories of knowledge may be distinguished to perform program supervision: 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. Most of this knowledge is modeled in two major concepts in our model, which are primitive or complex *operators*, and specialized *criteria*.

In addition to an abstract model of program supervision, we have developed a knowledge description language (named YAKL) to encode domain-specific information. YAKL is a model-theoretic approach to knowledge modeling in program supervision and allows the knowledge engineer to encode domain-specific program supervision knowledge bases. YAKL descriptions can be checked for consistency, and eventually translated into operational code. Some examples of the program supervision concepts described below are given using the YAKL syntax.

3.1 Supervision Operator

Supervision operators represent concrete programs (*Primitive operators*) or abstract processing (*Complex operators*). They both have input and output arguments (data or parameters). Both kinds of operators also encapsulate various criteria cr_k (which may be represented by rule bases) in order e.g. to manage their input parameter values (initialization criteria), to assess the correctness of their results (evaluation criteria on output data), and to react in case of bad results (repair criteria).

Primitive operators Referring to the formal definition of section 2, primitive operators are the rp_i representations of real programs p_i. The execution of a primitive operator corresponds to the execution of its associated program, provided that its execution conditions are true.

Complex operators They are the rc_j representations of higher level operations. They don't have attached operational actions but they decompose into more and more concrete (complex or

```
Primitive {
        name : o-muls
Functionality : thresholding
Argument descriptions:
Input Data
        Image name : e_image
                comment : "original image"
Input Parameters
        Float name : threshold
              default : 1
Output Data
        Image name : s_image
                comment : "thresholded image"
                I-O Relations :
                    s_image.path := e_image.path,
                    ....
Preconditions
        e_image.format == inrimage
        e_image.noise.kind == gaussian
Postconditions
        ...
Criteria omitted (see examples section 3.3)
....
Call
        syntax : cd e_image.path ";" muls -vs seuil s_image
}
```

Fig. 3. Example of a primitive operator o-muls. It performs a thresholding and has one input data (an image) and one numerical parameter (the threshold). Only the structural part is shown, criteria are detailed later. YAKL keywords are in bold face. Finally, the concrete syntax will be instantiated with the actual values of the input/output arguments for execution.

primitive) operators. Those decompositions are usually predefined by the expert in the knowledge base. The allowed 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 complexes ones. Since several operators can concretely realize one abstract functionality the specialisation decomposition type provides a way of grouping operators into semantical 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 specific operators. In a sequential decomposition some sub-operators may be optional. These decompositions –at different levels of abstraction– must end on primitive operators.

3.2 Arguments

Arguments are attributes of supervision operators. There are three sorts of arguments: input data, input parameters, and output data. Data arguments have fixed values which are set for input data like e.g., an input image, or computed for output data. The output data arguments can be

"assessed" during the reasoning by means of evaluation criteria. Parameter arguments are tunable, i.e. their values can be set by means of initialization criteria or modified by means of repair criteria.

3.3 Specialized Criteria

In program supervision different types of criteria are distinguished and their representations cr_k may be attached to supervision operators.

Common criteria For each operator an expert may define three kinds of criteria. Criteria provide a program supervision system with flexible reasoning facilities.

- *Initialization criteria* contain information on how to initialize values of input arguments.
- *Evaluation criteria* state the information on how to assess the quality of the actual results of the selected operator after its execution.
- *Repair criteria* express strategies of repair after a negative evaluation. A frequent repair strategy is simply to re-execute the current operator with modified parameter values. In complex operator they also express information propagation: e.g. the expert can express that the bad evaluation information has to be transmitted to a sub-operator, or to the father operator, or to any operator previously applied.

Criteria of complex operators For a complex operator an expert may define other specific criteria:

- *Choice criteria*: for a complex operator with a specialization decomposition type, choice criteria select, among all the available sub-operators, the operators which are the most pertinent, according to the data descriptions and the characteristics of the operators. This kind of criteria is used for planning purposes.
  Here is an example of a choice rule in YAKL language:

```
Choice criteria
Rule name : r-choice
comment :  "choice of operator constr-ch-with-filter if image is noisy"
    Let ?c a Context
    If
      ?c.noise == present,
      size-filter > 0
    Then
      use-operator constr-ch-with-filter
;
```

- *Optionality criteria*: for a complex operator with a sequential decomposition type, optionality criteria decide if an optional sub-operator has to be applied depending on the dynamic state of the current data. Such criteria increase the flexibility of the system.

operator	: complex-operator | primitive-operator
primitive-operator	: common-part call
complex-operator	: common-part body
common-part	: identification arguments common-criteria preconditions postconditions effects
arguments	: input-data input-parameters output-data
common-criteria	: initialization-criteria evaluation-criteria repair-criteria
body	: decomposition complex-criteria data-flow
complex-criteria	: choice-criteria | optionality-criteria
decomposition	: decomposition-type sub-operators
decomposition-type	: specialization | sequence

Fig. 4. Model grammar: a BNF rule is represented by "*left-part* : *right-part*", where *left-part* is a non terminal that is expanded into all *right-part* components. A pipe represents a "or", e.g. the first rule means that an *operator* can be expanded either as a *complex-operator* or as a *primitive-operator*. A blank represents a "and", e.g. the second rule means that a *primitive-operator* is expanded as a *common-part* and a calling interface (*call*).

3.4 Knowledge Base Concept Overview

In this paragraph we summarize the relationships between the main concepts of a program supervision knowledge base. Figure 3 shows an abstract view of knowledge base concepts in the form of a (BNF) grammar.

The identification of an operator contains its name, functionality and possibly a comment and a list of symbolic characteristics. The calling interface of a primitive operator describes all the information needed for the effective execution of the code. Pre- and post-conditions are tests which have to be checked before and after the execution of the operator. Effects are statements which are established after the execution of the operator.

4 Program Supervision Reasoning

In this section we briefly present how we model the reasoning process for program supervision into several phases. A program supervision task is solved by a problem-solving method adapted to experts' reasoning process and domain requirements.

4.1 Reasoning Phases

The reasoning of a program supervision is implemented in the supervision *engine*. The role of the engine is to exploit knowledge about programs in order to produce a plan of programs, that achieves the user's goal. It emulates the strategy of an expert in the use of programs. The final plan that produces satisfactory outputs is usually not straightforward, it often results from several trials and errors. The reasoning engine explores the different possibilities and computes the best one, with respect to expert criteria, available in the knowledge base.

The engine roughly automates a cycle of four reasoning phases to solve user's problem (see left part of figure 5), that can be completely or only partly automated.

– The `planning phase` can use a hierarchical strategy or a mechanism based on preconditions and effects of available programs on data description.
– The `execution phase` runs (directly or via a communication protocol) the programs with current data and adequate parameter values.
– The `evaluation` of the returned results can sometimes be fully automated thanks to measurement computation, expert-defined criteria, methods to compare results with reference cases, etc.
– Finally,the `repair phase` offers smart backtrack possibilities to different return points, depending on the type of decisions that can be reconsidered.

According to the type of reasoning process to carry out, there may exist variants of the four phases of this general model, that can also be more or less interleaved. For instance the planning phase may be based on a Hierarchical Task Network (HTN) or may use Partial Order Planning (POP).

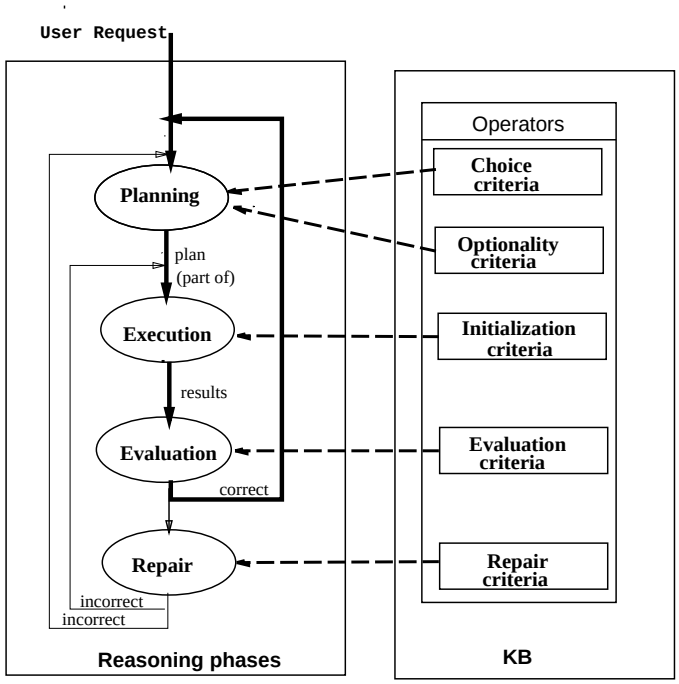

Fig. 5. A user request is solved by different reasoning phases: (1) the `planning` of (part of) the solution, that selects and organizes programs into a sequence of actions, (2) the monitored `execution` of each action after appropriate refinement and parameter tuning, (3) the `evaluation` of the results and finally (4) the `repair` of the (partial) solution if results are not satisfying. Bold arrows show the main recursive loop. Plain arrows shows the repair loops; repair is performed either by simple re-execution of the same operator or by replanning. Dotted arrows show which type of knowledge base (KB) criteria is used for each reasoning phase.

4.2 Role of Criteria

During the reasoning, the engine not only exploit the operator descriptions, but also criteria cr_k that are described in the knowledge base. The specialized criteria play different roles and they are involved in different phases of the program supervision reasoning. For example, choice criteria between supervision operators are related to the *planning* phase, while initialization criteria are related to the *execution* phase. Figure 5 (right part) sumerizes the relationships between these specialized criteria and the different phases of the reasoning. The richer the knowledge base is in terms of criteria, the more flexible the related reasoning phase will be.

5 Re-using a Set of Programs with Program Supervision

Program supervision aims at large scale software management and utilization. The main phases of planning, execution, evaluation, and repair are generic tasks of program supervision, software reuse and software engineering methodology. Specific to program supervision is the knowledge-based approach. Different categories of knowledge must be explicited in order to adequately build a program supervision system.

5.1 Requirements for Using Program Supervision techniques

In this section we analyze the properties the programs p_i and their arguments must verify in order to be candidates for re-use with program supervision techniques. In addition, we propose general advices about the construction of operators (rp_i and rc_j) and criteria (cr_k).

Program properties If there is only one unique and clearly defined functionality for each program p_i the model is directly applicable. A primitive operator rp_i is thus created for each p_i. 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 rp_i 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. This of course implies an additional effort at first, but in the long term it is an advantage both for code maintenance and for knowledge base evolution.

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.

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, program supervision implies to make explicit for each program its internal parameters by rewriting it 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 p_i 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 rp_i describing p_i. When parameters exist, the operator rp_i describing p_i must contain the knowledge on how to tune them. This is relatively easy for initialization criteria, but is more demanding for repair criteria expressing how to adjust their values w.r.t. bad result evaluations.

Complex 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 complex operator per functionality. The type of decomposition of this complex operator is of specialization type and the sub-operators are the alternative primitive operators. When typical program combinations c_j 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 complex operator (rc_j) per typical combinations c_j. For instance if the combination c_j is a sequence of programs, the type of decomposition of the complex operator is sequence and the sub-operators are the ordered list of the primitive operators rp_i representing the programs p_i in the sequence c_j. The knowledge base can contain several abstraction levels when the body of a complex operator is itself composed of other complex operators. It is thus possible to represent taxonomies of functionalities with complex operators. If for one single functionality, there exist several operators that achieve it, the existence of these alternatives leads to a richer and more flexible knowledge base, with a wider range of applicability.

Criteria properties First, the criteria are specialized: if there are known criteria for choice of sub-operators, input parameters initialization, output data evaluation, or repair strategies, even if these criteria are only available for specific applications, they can easily be expressed in the knowledge base using the adequate criteria type provided by the model. However, the criteria are not mandatory: 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.

Finally, the criteria can manage the degree of interactivity with the user. If there exist methods for automating the computing of values (parameter initialization methods, parameters adjustment methods, or methods for the evaluation of the results), these methods can be directly translated into specialized criteria. If these methods do not exist, specialized 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.

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 methology of knowledge base building: the easiest way is to begin by describing concrete individual programs, then to create higher levels of abstraction using complex operators. Criteria may be added afterwards, the more criteria the knowledge base contains the more efficient and flexible the program supervision process will be.

5.2 Program Supervision Knowledge Bases

This section shows on three very different examples how program properties have influenced the knowledge base building. We have developed two program supervision engines compatible with the knowledge representation model presented in section 3. For more details on these engines see [3, 14] for the Ocapi engine and see [16] for the new Pegase one. These engines have been used for building several knowledge bases.

Progal The first example is an application in astronomy, where the role of the program supervision system is to automate a complete processing in order to cope with possible variations in the input data (images of galaxies) [11, 17]. There was already a modular set of 37 image processing programs. Only one program has been split into two programs p_i, to allow an easier use of the repair knowledge. Thus, in the Progal knowledge base 38 primitive operators rp$i$ have been created. Progal is a rich knowledge base with 54 complex operators rc_j and a lot of abstraction levels. The criteria cr_k are numerous and fully automatic: there are 20 choices between operators, 16 parameter initializations, 11 result evaluations, 21 repair criteria. Thanks to all those criteria, the complete image processing for morphological galaxy description is fully automated and directly provides inputs for an automatic galaxy classification system. A first version of the knowledge base has been developed with the Ocapi engine, the current knowledge base works with the Pegase engine.

Promethee The second example is a stereo-vision-based module performing few technical functionalities, as 3D computing and obstacle detection. This module is difficult to use due to the existence of a lot of technical parameters to tune. The Promethee knowledge base [12] contains 24 primitive operators corresponding to the 24 programs and only 15 complex operators. Among the 120 criteria there are only 15 choice criteria. Some of the 20 evaluation criteria are interactive ones, because of the generality of this module which can be applied on very different images corresponding to various application domains. On the other hand, the technical knowledge on how to initialize and adjust the parameters is important thanks to 64 initialization criteria and 21 repair criteria. This knowledge base works with the Ocapi engine.

FAMIS The third example falls in the domain of medical imaging. The objective is to offer clinicians a wider access to evolving medical image processing techniques and more precisely with Factor Analysis of Medical Image Sequences (FAMIS [5, 1]). In this case, there is no need for complete automation. There are few (8) big programs performing several functionalities, plus 7 secondary programs. So, the knowledge base [4] contains more primitive operators (22 rp_i) than there exist actual programs (15). There are 11 complex operators with alternative or sequential arrangement decompositions. The criteria are numerous in the current knowledge base : 120 criteria among which 10 choice criteria, 3 optionality criteria, 30 parameter initializations, 30 results evaluations, 50 repair criteria. Most of the evaluations criteria and some of the choice and optionality criteria work in interaction with the clinician end-user. This knowledge base works with the Pegase engine.

6 Conclusions

We can summarize what program supervision can do for program re-use from the points of view of the code developers, the experts and the end-users. Obviously, code developers can not expect any improvement in the quality, complexity or speed of the individual programs only by application of program supervision techniques. A first major impact of program supervision for code developers is that the building of a program supervision knowledge base may influence the methodology of code design, leading to more modular and structured codes. In spite of this additional coding effort, it contributes in the long term to better code maintenance and knowledge base evolutions. In fact the program supervision approach provides the resulting system with extensibility, as it is easier to add new programs in the library. The second major impact for code developers concerns the diffusion of their code. The knowledge encapsulation of the codes using program supervision

techniques allows a wider usage of the codes because the program supervision system can adapt dynamically its behavior to the end-user data.

For the expert we propose a model which provides a framework and a clear description of the structure of the knowledge involved in program supervision. Yet, even if all the requirements on programs previously mentioned in section 5.1 are met, the building of a knowledge base for program supervision is still a big effort for the expert. A partial help can be provided by verification techniques as outlined in [8]. However, once the program supervision knowledge base has been written following the advices developed in section 5.1, the managing of the set of programs is completely handled by the program supervision engine. No more effort is necessary for the expert, because the knowledge modeling is completed. This knowledge capitalization is an important result per se; it can cope the fact that the code developers or experts are not numerous enough or are no longer available.

For an end-user, program supervision techniques facilitate the re-use of a set of programs. The end-user has only to provide a request with input data, and possibly some constraints on output data and some intermediate result evaluations. The program supervision approach provides the resulting system with more or less autonomy, since the user is not burdened with technical processing problems. In certain cases completely autonomous systems can even be developed.

References

1. H. Benali, I. Buvat, and al. A statistical method for the determination of the optimal metric in factor analysis of medical image sequence (FAMIS). *Physics in Medicine and Biology*, 38:1065–1080, 1993.
2. British-Aerospace. "VIDIMUS Esprit Project Annual Report". Technical report, Sowerby Research Centre, Bristol, England,, 1991.
3. V. Clément and M. Thonnat. Integration of Image Processing procedures, Ocapi: a Knowledge-Based Approach. *Computer Vision Graphics and Image Processing: Image Understanding*, 57(2), March 1993.
4. 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'97*, volume 3035, February 1997.
5. F. Frouin, J.P. Bazin, M. Di Paola, O. Jolivet, and R. Di Paola. Famis : A Software Package for Functional Feature Extraction from Biomedical Multidimensional Images. *Computerized Medical Imaging and Graphics*, 16(2):81–91, 1992.
6. M. Haest et al. ESPION: An Expert System for System Identification. *Automatica*, 26(1):85–95, 1990.
7. J.E. Larsson and P. Persson. An Expert System Interface for an Identification Program. *Automatica*, 27(6):919–930, 1991.
8. M. Marcos, S. Moisan, and A. P. del Pobil. A Model-Based Approach to the Verification of Program Supervision Systems. In *4th European Symposium on the Validation and Verification of Knowledge Based Systems*, pages 231–241, June 1997.
9. S.H. Nawab and V. Lesser. Integrated Processing and Understanding of Signals. In A.V.Oppenheim and S.H.Nawab, editors, *Symbolic and Knowledge-Based Signal Processing*, pages 251–285. Prentice Hall, 1992.
10. C. Shekhar, S. Moisan, and M. Thonnat. Towards an Intelligent Problem-Solving Environment for Signal Processing. *Mathematics and Computers in Simulation*, 36:347–359, March 1994.
11. M. Thonnat, V. Clement, and J.C. Ossola. Automatic galaxy classification. *Astrophysical Letters and Communication*, 31(1-6):65–72, 1995.
12. M. Thonnat, V. Clement, and J. van den Elst. Supervision of perception tasks for autonomous systems: the OCAPI approach. *Journal of Information Science and Technology*, 3(2):140–163, Jan 1994. Also in Rapport de Recherche 2000, 1993, INRIA Sophia Antipolis.

13. M. Thonnat and S. Moisan. Knowledge-based systems for program supervision. In *First international workshop on Knowledge-Based systems for the (re)Use of Programs libraries KBUP'95*, pages 4–8, Sophia Antipolis, France, March 1995. INRIA.
14. J. van den Elst, F. van Harmelen, G. Schreiber, and M. Thonnat. A functional specification of reusing software components. In *Sixth International Conference on Software Engineering and Knowledge Engineering*, pages 374–381. Knowledge Systems Institute, June 1994.
15. 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.
16. R. Vincent and M. Thonnat. Planning, executing, controlling and replanning for ip program library. In *Proc. of 8th Artificial intelligence and Soft computing ASC'97*, July 1997.
17. R. Vincent, M. Thonnat, and J.C. Ossola. Program supervision for automatic galaxy classification. In *Proc. of the International Conference on Imaging Science, Systems, and Technology CISST'97*, June 1997.