Escuela Superior de Tecnología y Ciencias Experimentales

Departamento de Informática

Verification and Validation of Knowledge-Based Program Supervision Systems / Verificación y Validación de Sistemas de Supervisión de Programas Basados en el Conocimiento

TESIS DOCTORAL

presentada por

María del Mar MARCOS LÓPEZ

dirigida por

Sabine MOISAN
Angel PASCUAL DEL POBIL Y FERRÉ

Castellón, 22 de Febrero de 1999

a Carmen

Acknowledgements

This thesis is the result of several years of work and close collaboration between the ORION research team at INRIA–Sophia Antipolis (France) and the Robotic Intelligence Lab. of Jaume I University in Castellón (Spain). I would like to express here my sincere gratitude to all the people and institutions that have made this adventure possible.

To Monique Thonnat, who accepted me within ORION, proposed a fascinating thesis subject, and bet on a satellite student in spite of the complications that long-distance work implies. To my supervisors Sabine Moisan and Angel P. del Pobil, who helped me to accomplish this work, always open to discussing new ideas and attentive to the electronic mail, fax or telephone. To Frank van Harmelen, with whom I have had enlightening and encouraging discussions from the beginning of my work.

I am grateful to Jaume I University for granting me the mobility necessary to accomplish this project, and for facilitating the writing and defence of my thesis in the English language. I am also grateful for all the human and material resources that both Jaume I University and INRIA–Sophia Antipolis have placed at my disposal.

Other two universities have made important contributions for the realization of my thesis. I am grateful to Prof. Treur from the Mathematics and Computer Science Department of the Free University of Amsterdam for inviting us for a visit during which we came in contact with the KIV group. I am also grateful to Prof. Menzel from the Institut für Logik, Komplexität und Deduktionssysteme of Karlsruhe University for receiving us for the KIV course soon afterwards, and for facilitating the installation and utilisation of the tool in my university. I also want to thank Arno Schönegge for the speed and clarity of his answers to the problems I encountered using KIV.

I should also mention all the institutions that have contributed to the financial support:

- The ECC COMETT Programme.
- The Fundació Caixa Castelló.
- The Picasso Integrated Actions between Spain and France.
- The Spanish Ministry of Education and Culture.

My working environment during these last years has been excellent, in both Castellón and Sophia. At the university, my colleagues from the Computer Science Department always helped me so that everything went smoothly. At INRIA, the people I met during the six months I spent there every year made me feel at home. With the program supervision group —formed by Sabine Moisan, John van den Elst, Régis Vincent and Monica Crubézy— I learned the importance of teamwork. Our discussions and interactions have been the culture medium for many of the ideas in this thesis. I owe this group, and the rest of the members of ORION, my passion for Artificial Intelligence. Specially, I am grateful to John and Monica for discovering, amongst other things, that we can —and should!— have fun while working seriously.

And last, but not least, I would like to give my warmest thanks to my wholehearted fans for their support (they will recognize themselves while reading this). I really don't know how I would have succeeded without their love and encouragement. Specially, I want to give my thanks to my mother Carmen, my brother Jose and my sister Raquel, to Tico, to Javier and Empar, and to Divi, Nando and Cristina.

Thank you all!

Mar Marcos
February 1999.

Contents

Chapter 1

Introduction

This thesis is devoted to the verification and validation of knowledge-based program supervision systems. In this chapter we first present the knowledge-based program supervision techniques. Afterwards we justify the interest of the verification and validation of these systems. Then we introduce the framework for the design and implementation of knowledge-based program supervision systems in our team, which has influenced our approach to their verification and validation. Finally we outline the structure of this thesis.

1.1 Program supervision

Research in many fields —like scientific computing or image processing— provides us with libraries of programs implementing up-to-date techniques. Although *availability* is a key issue for a wide utilisation of these techniques, the *ease of use* is not less important. Potential users may find that programs are difficult to handle. The sources of difficulties reside in characteristics inherent, not to the programs themselves, but to their application to a concrete problem [Mili, 1995]. These characteristics range from "external" properties (e.g. in which situations should we use this program?) to "internal" functioning details (e.g. how should we initially set its parameters? or how sensible are the results to missing data?). Users find the *ad-hoc* utilisation of two or more programs for solving a complex problem even more difficult. Generic on-line helps of programs, though valuable, are not sufficient because they cannot replace the extensive expertise on the use of programs that is necessary to solve different or even new problems.

Different streams of research arise with the purpose of giving support to unexperienced users in the utilisation of program libraries. Some research focuses on documentation techniques allowing users to assess the applicability of programs to the problem at hand, and thus indirectly supporting program selection [Mili, 1995]. Other research concentrates on the automation of this and other activities, e.g. tuning of program parameters. Several examples of the latter in the field of image processing are presented in [Matsuyama, 1989].

We use the term *program supervision* (PS) to denote those techniques aiming at the automation of (some of) the activities involved in the skilled use of a program library. Given an intended processing goal, PS systems have to deal with the selection of the appropriate programs, their scheduling, and their execution, among other problems. PS systems differ in the number of activities they cover [Thonnat and Moisan, 1995]. Furthermore PS systems differ in their degree of automation, ranging from highly interactive systems to fully automated ones. In general, a high degree of automation can only be attained by narrowing the application domain.

1.2 Knowledge-based program supervision systems

To accomplish their task, PS systems need the expertise involved in the use of programs. Two alternatives exist for embodying this expertise, namely, its integration in the code of programs or its separated explicit representation. The advantages of making this knowledge explicit are numerous. For instance, it allows the separation of the programs themselves from the control strategies that command their use. Programs designed in this way can therefore accommodate a variety of control strategies [Crevier, 1993].

Knowledge-based techniques provide an appropriate ground for the implementation of PS systems. On the one hand, they allow the explicit representation of knowledge. On the other, they provide for this purpose a set of knowledge representation formalisms that are suitable for the implementation of PS knowledge, e.g. production rules. Henceforth, when we name PS systems we will refer to knowledge-based PS systems.

In the literature we find many implementations of PS systems with a knowledge-based architecture, mainly in the field of image processing. These systems separate the programs to supervise from the knowledge necessary to do it. This knowledge can be of different types, e.g. control strategies or specific objects manipulated. At the same time, these different types of knowledge are more or less specific to the application domain, e.g. manipulated objects

are inherently domain-specific. Finally, most of the systems in the literature use planning techniques to tackle the PS problem.

Early systems —reviewed in [Matsuyama, 1989]— comprise consultation systems for program selection and parameter tuning, and composition systems for the combination of library programs into more complex ones. Several PS systems are dedicated to particular application domains. Examples are: MVP [Chien, 1994] which generates executable scripts for the analysis of interplanetary images; VISIPLAN [Gong and Kulikowski, 1994], [Gong and Kulikowski, 1995] which is devoted to the generation of processes for the segmentation and recognition of biomedical magnetic resonance images; and VSDE [Bodington, 1995] which automates the configuration of image processing systems for the inspection of defects in industrial products.

Other work searches to give more general PS support, i.e. reusable across different application domains. As examples we can cite OCAPI [Clément and Thonnat, 1993], [Thonnat et al., 1994], BORG [Clouard et al., 1993], and SCARP [Willamowski et al., 1994]. All three examples are domain-independent shells or engines. The development of a PS system for a particular application requires adding a domain-specific knowledge base.

1.3 Verification and validation needs of program supervision systems

As knowledge-based systems become a standard in software development, the interest in verification and validation techniques adapted to their particularities has grown. Knowledge-based systems cannot be verified against a complete requirement specification, as is the case of traditional software, because of the problems that they solve. These problems are difficult or ill-defined, and usually lack such kind of specifications. Knowledge-based systems, at the same time, represent and manipulate knowledge explicitly. This knowledge is often represented by means of rules which permit us to check the logical consistency and completeness of the knowledge base implementation. The verification and validation without requirement specification, on the one hand, and its focus on rule-based implementations, on the other, characterise most of the existing work on the verification and validation of knowledge-based systems.

The development process of PS systems demands the utilisation of verification and validation techniques in order to become reliable industrial applications. This is especially important

in the case of applications where human interaction does not exist, i.e. in automated PS systems. An example is PLANETE [Shekhar et al., 1995], which is a real-time PS system for the supervision of perception programs in the framework of an on-board guidance system.

All presented PS systems and shells are based on the different types of knowledge necessary to perform the PS related activities. These types of knowledge are implemented using specialised knowledge representation formalisms in addition to production rules. For instance, PS knowledge usually includes descriptions of programs at different levels of abstraction, which are often implemented using object-oriented representations. PS knowledge often includes production rules, e.g. to set the initial value of program parameters. The use of different types of knowledge implemented using multiple representation formalisms makes the construction and maintenance of PS knowledge bases difficult [Bodington, 1995], [Chien, 1996]. PS systems need therefore verification and validation techniques to deal with these characteristics.

While considerable research effort has focused on the verification and validation of rule-based systems, little work has concentrated on other representation formalisms. What is more important, the verification and validation of different types of knowledge in the sense explained above has not been considered either. In this thesis we deal with techniques adapted to the verification and validation of PS systems. The systems that have been the target of our study are the PS systems developed within the ORION research team at INRIA–Sophia Antipolis.

1.4 Framework for the verification and validation of program supervision systems

Herein we introduce the aspects that constitute the framework for the design and implementation of PS systems in our team. Our PS systems support all the activities involved in the use of a program library, namely the selection of programs, their scheduling, their execution, the evaluation of the execution results, and the consequent correction on the previously taken decisions if the results are deemed unacceptable.

PS systems embody the expertise involved in the use of programs in a knowledge-based architecture. The knowledge-based **architecture of PS systems** in our team consists of a PS engine, a PS knowledge base encapsulating the expertise on the use of programs, and the library itself. We shall consider that the PS knowledge base constitutes the domain-specific part of the application and that the PS engine is the domain-independent one.

From previous experiences in the development of PS systems in our team (mainly the different applications built from OCAPI engine), a **conceptual model of PS** has been drawn. This conceptual model includes a series of knowledge concepts, which are interrelated in a complex organisation, and a reasoning strategy. The knowledge concepts correspond to the different types of knowledge employed to perform the activities mentioned above and the reasoning strategy comprises the steps necessary for doing it. Each step makes use of precise knowledge concepts in performing its task. These knowledge concepts and reasoning strategy are the most commonly used in PS according to our view. They serve as a basis for the implementation of PS knowledge bases and engines.

In addition, as a result of the previous PS experiences, the interest of the adaptation of the reasoning strategy in the conceptual model to the particularities of new applications has been confirmed. The LAMA **platform for knowledge-based system development** is intended to facilitate the adaptation of reasoning strategies, i.e. engines. LAMA is at present devoted to the development of PS systems. For this purpose it provides a library of components for the construction of PS engines and knowledge bases. With the object of facilitating PS system development, these components lie at a level of abstraction which is more or less close to the PS conceptual model while hiding implementation details. The PS systems that are currently being developed in our team are implemented under the LAMA platform.

In the light of the previous aspects, we focus on the verification and validation of PS knowledge bases intended for new applications, and, to some extent, on the verification and validation of PS engines designed to meet particular reasoning strategies. In the verification and validation of knowledge bases, we exploit the PS conceptual model which reflects the necessary knowledge concepts, their interrelations, and the precise way in which they are used to accomplish the PS activities. Finally, we search the integration of the adequate verification and validation techniques/methodologies in the development process of PS systems using LAMA.

1.5 Structure of the thesis

This thesis is structured as follows. In chapter 2 we review the state-of-the-art on the verification and validation of knowledge-based systems.

In chapter 3 we describe in depth the framework for the design and implementation of PS systems, especially the conceptual model of PS and the LAMA development platform.

Regarding the PS conceptual model, we give some examples of how it can be exploited in the verification and validation of PS knowledge bases. We present in chapter 4 the approach that has been adopted to put our ideas into practice.

Our approach to the verification and validation of PS knowledge bases, which exploits the PS conceptual model, is structured around the realizations presented in the following two chapters. In chapter 5 we present a knowledge modeling of different PS systems which is intended to enhance our understanding of the different knowledge concepts and their use in PS reasoning.

In chapter 6 we use the descriptions in the knowledge modeling to define the properties that PS knowledge bases should verify. Then we describe the new LAMA module that we have implemented to verify them, which uses currently available verification and validation techniques adequate to the target representation formalism in LAMA.

Regarding the verification and validation of PS engines, in chapter 7 we report on some experiences to assess the feasibility of the use of software engineering verification techniques in the development of PS systems.

Finally in chapter 8 we summarise the main contributions of this thesis, discuss their limitations, and outline some future prospects.

Several parts of this thesis have been published or accepted for publication:

- A summary of the state-of-the-art (in chapter 2) is published in:
  Verification and validation of knowledge-based program supervision systems.
  M. Marcos, S. Moisan, and A. P. del Pobil
  IEEE International Conference on Systems, Man and Cybernetics (SMC-95), October 1995, Vancouver, Canada.
- An overview on our approach to the verification and validation of PS systems (in chapter 4) is published in:
  A Model-Based Approach to the Verification of Program Supervision Systems.
  M. Marcos, S. Moisan, and A. P. del Pobil
  Fourth European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV-97), June 1997, Leuven, Belgium.
- Part of the knowledge model of PS systems (in chapter 5) is published in:
  Knowledge Modeling of Program Supervision Task.

M. Marcos, S. Moisan, and A. P. del Pobil
Eleventh International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA-98-AIE), June 1998, Benicàssim, Spain.

- Part of the knowledge model of PS systems and some considerations on the future prospects of verification and validation in the frame of LAMA (chapters 5 and 8) are published in:
  Experiments in Building Program Supervision Engines from Reusable Components.
  M. Crubézy, M. Marcos, and S. Moisan
  Workshop on Applications of Ontologies and Problem-Solving Methods, Thirteenth European Conference on Artificial Intelligence (ECAI-98), August 1998, Brighton, England.
- The knowledge model of PS systems and its application to the verification of PS knowledge bases (chapters 5 and 6) has been accepted for publication in:
  Knowledge Modeling of Program Supervision Task and its Application to Knowledge Base Verification.
  M. Marcos, S. Moisan, and A. P. del Pobil
  Applied Intelligence.

Chapter 2

State-of-the-art on Verification and Validation of Knowledge-Based Systems

With greater acceptance of knowledge-based systems, the interest in verification and validation techniques adapted to their characteristics has grown. In this chapter we aim at providing an overview on the field of verification and validation of knowledge-based systems.

The chapter is structured as follows. We first present the terminology in the field and the characteristics of the existing work. Then we focus on the work on the verification of knowledge base anomalies. Finally we discuss the limitations of this approach and conclude.

2.1 Verification and validation terminology

Under the name *verification and validation* (V&V) we find a set of activities aiming at assuring a certain degree of quality and reliability in knowledge-based systems. The terms verification and validation have sometimes been used without distinction in the knowledge-based system field. Even more, there is no agreement on the precise activities that each involve. Without the aim of being exhaustive, we present next different points of view that can be found in the literature.

The generic term validation is used in [López et al., 1990]. It is mainly concerned with the correct structure and adequate behaviour of the knowledge-based system, and comprises the following activities:

- structural verification, which consists in checking the knowledge base (KB) for a set of anomalies which are tightly related to the knowledge representation formalism.
- functional verification, which aims at assuring that the output of the knowledge-based system complies with the semantics of the real world. This implies both the validation of the inference engine and the validation of the KB contents with respect to the semantics of the domain (knowledge refinement).
- evaluation, which aims at measuring the performance of the knowledge-based system based on structural and functional characteristics.

Validation is also used as generic term in [Laurent, 1992]. The distinction among the different validation activities is made according to the kind of specifications that they involve:

- objective validation or verification, for those processes based on formal specifications.
- interpretative validation or evaluation, for processes based on semi-formal specifications.

[Gupta, 1993] follows the terminology in software engineering and distinguishes validation from verification. Validation is the process of determining whether the knowledge-based system complies with the user's requirements, whereas verification is the process by which the requirement specification, design and implementation are checked for consistency. Verification includes the process of anomaly detection in the KB based on the logics of the knowledge representation. [Batarekh et al., 1991] and [Meseguer and Preece, 1995] make a similar distinction regarding the main V&V issues.

Henceforth we will refer to the terms verification, validation and test. In our view, verification is concerned with the internal consistency of specification, design and implementation, i.e. correctness of the system at all levels, but also with the compliance of each stage with respect to the others. Validation involves determining the compliance of the system with respect to the (implicit or explicit) user's requirements. Finally testing, which is a widely acknowledged method of doing validation, consists in checking the system correctness by executing it on sample data sets.

2.2 Verification and validation of knowledge-based systems

One characteristic of knowledge-based systems, which distinguishes them from traditional software, is that they solve difficult or ill-defined problems for which no general efficient, algorithmic solution exists. In consequence knowledge-based systems lack a precise requirement specification. Typically, development proceeds in an evolutionary manner –usually via prototyping– until a system that satisfies the user's implicit requirements is produced [Meseguer and Preece, 1995]. Although current development methods are much more structured, prototyping is often mentioned in the literature.

In the absence of a detailed requirement specification, V&V originally involved comparisons of knowledge-based system performance against human expert's performance. For the same reason, most of the early work concerns V&V techniques that do not depend on the existence of a specification of requirements [Meseguer and Preece, 1995]. An example is the work done on the detection of implementation-dependent anomalies in the KB.

Verification by detection of implementation-dependent anomalies in the KB and testing are the major V&V techniques in many systems [Prerau et al., 1993], [Meseguer and Preece, 1995]. One problem with testing is that it is not systematic [Rousset, 1988]. Among other reasons, this is due to the difficulty of determining each path through the system [Preece et al., 1992]. Moreover testing delays V&V until the end of implementation. On the other hand, KB anomaly detection involves checking for certain anomalies that may not be revealed through a test phase, and it is therefore unavoidable [Cragun and Steudel, 1987], [Meseguer and Preece, 1994]. It is also important since it can —and should— be performed before the system is fully functional.

Next we review some of the numerous bibliographical references on the analysis of KB implementation-dependent anomalies and examine their salient characteristics.

2.3 Verification by detection of implementation-dependent anomalies in the knowledge base

As mentioned before, most of the past V&V work consists in verification by checking the KB for certain anomalies which could indicate errors in the construction of the system. Typical anomalies are inconsistency, incompleteness and redundancy. These anomalies are defined

in terms of the used knowledge representation formalism, mostly rule-based, and the target deduction model. A formulation of the most common rule-based anomalies can be found in appendix A.

It must be noticed that anomalies are not errors but rather symptoms of probable errors in the KB [Preece and Shinghal, 1992]. For instance, the redundancy anomaly, which can be detected in rule bases by checking for duplicate rules, may indicate a simple editing error or a case in which one or both rules are incorrect.

The techniques below have been grouped according to how they approach the definition of KB anomalies.

2.3.1 Rule syntax-based definitions

The following techniques define KB anomalies using exclusively rule syntax. They check for anomalies by means of pairwise comparisons of rules. For instance, to check for inconsistency in propositional logic rules [Meseguer and Preece, 1994]:

> Given the semantic constraint $C(x, y)$ expressing two inconsistent facts,
> and the rules r_i and r_j,
> with consequents $conseq(r_i) = x$ and $conseq(r_j) = y$,
> and antecedents $antec(r_i)$ and $antec(r_j)$,
> If $antec(r_i) \subseteq antec(r_j)$ then inconsistency.

The only semantic constraints that these techniques consider are those related to the representation formalism, or logical constraints, e.g. a fact and its negation or two different values for a singled-valued attribute.

The completeness and consistency checker of the ONCOCIN KB [Suwa et al., 1982] is the earliest work. The ONCOCIN system is an EMYCIN-like knowledge-based system. A rule in ONCOCIN has an action part which concludes a value for some parameter from the values of other parameters in the condition part. The rule also specifies the context in which it applies. The ONCOCIN's rule checker points out inconsistent, redundant, subsumed, and missing rules.

CHECK [Nguyen et al., 1985] is the completeness and consistency checker for KBs in the LES framework. Besides the problems indicated by the previous checker, CHECK signals

circular rule chains, dead-end clauses and unreachable clauses. A further version of CHECK [Nguyen et al., 1987] includes checks for unnecessary conditions and unreferenced and illegal attribute values.

ARC [Nguyen, 1987] is the rule checker of the ART framework. In addition to CHECK, ARC detects unnecessary conditions in antecedents and signals redundant, subsumed, and conflicting rule chains.

2.3.2 Deduction-based definitions

The following techniques make use of definitions that take into account the whole set of deductions in the KB. Except for the first reference, the rest are based on ATMS notions of environment and label [de Kleer, 1986]. An environment for a fact x, $E_i(x)$, is a minimal conjunction of external facts, or initial fact base, supporting it. The label $L(x)$ is the disjunction of all the environments for x and thus it represents all the elemental ways to deduce the fact.

Then, a rule base is inconsistent if an integrity constraint can be deduced from some legal input set. To check for inconsistency [Meseguer and Preece, 1994]:

> Given the semantic constraint $C(x, y)$ expressing two inconsistent facts,
> and the environments $E_i(x)$ and $E_j(y)$,
> If $E_i(x) \wedge E_j(y)$ then inconsistency.

Besides logical constraints, these techniques often consider domain-dependent ones, or expert constraints, e.g. $man(x) \wedge pregnant(x)$. These constraints serve to introduce the notion of semantic inconsistency of facts even when they are logically consistent.

The definition of consistency used in GCE [Beauvieux and Dague, 1988], [Beauvieux and Dague, 1990] is based on both logical constraints and expert constraints, and considers propositional logic and Attribute-Object-Value (AOV) rules. The consistency check is performed using the base models of the KB, which are the different maximal consistent fact bases of the KB. These base models are built along with the KB, and are employed after each KB modification (addition or retraction of a rule or a constraint) to determine whether there is an inconsistency as a result.

COVADIS [Rousset, 1988] checks a KB for inconsistency. COVADIS is intended to be applied to KBs written in the MORSE framework. MORSE rules are Attribute-Value (AV)

rules. The idea is generating from the KB the specification of all initial fact bases from which an inconsistency can be deduced. This specification will be submitted to the expert, who will decide on whether they are meaningful or not. The procedure uses the notion of context of a fact, which similarly to the label notion constitutes a specification of all initial fact bases justifying it.

KB-Reducer [Ginsberg, 1988] checks a KB for redundancy and inconsistency. KB-Reducer builds from the KB all possible initial fact bases from which a fact can be deduced or labels. It orders the rules into levels according to a "depends on" relation. Roughly speaking, a rule r_i depends on a rule r_j if and only if r_j asserts a literal that appears in the antecedents of r_i. Then, KB-Reducer treats rules in this order, updating the partial labels of their conclusions consequently. Checks for redundancy and consistency are performed after processing every rule by using the partial labels. For instance, to check for inconsistency, a subset/superset test is made between the partial labels of the conflicting hypothesis involved.

[Meseguer, 1992] presents a similar procedure to check modular rule bases for redundancy, inconsistency, circularity, and useless objects. Checks are performed by testing relations among environments and labels. After every KB modification, environments and labels of KB objects are computed, and relations among them are tested. The procedure repeats only the test of KB objects which labels change from the verified KB to the new one.

[Loiseau, 1992] presents COCO which is a system to check the consistency of KBs written in a restriction of first order logic (first order production rules without functional symbols). It makes use of an explicit distinction between indisputable and revisable knowledge for the definition of consistency. A KB is said to be inconsistent if and only if there exists one initial meaningful fact base from which an inconsistency can be deduced. Initial meaningful fact bases are those from which inconsistencies cannot be deduced by using only indisputable knowledge. They are also modeled as labels.

2.3.3 Translation-based definitions

Different techniques translate the KB into structures on which the anomalies are reinterpreted. These techniques often exploit algorithms specific to the structures in the anomaly detection procedures. Besides, the target structures usually enhance the readability of the KB, and therefore are also useful for manual V&V by inspection.

2.3.3.1 Decision table-based definitions

[Puuronen, 1987] presents a decision table-based technique to check ONCOCIN-like rule bases. The KB is mapped onto modified decision tables on which inconsistency, redundancy and incompleteness are checked. The algorithms proposed for the different checks are based on the cardinal numbers of rules, which specify the different cases that the rule covers.

ESC [Cragun and Steudel, 1987] implements another decision table-based technique dealing with AV-like KBs. ESC maps the KB onto a master decision table and then splits it into disjoint subtables, each of them corresponding to a set of context related rules that are checked together. First ESC checks rules that cover non-disjoint cases and signals some redundancy problems. Then it performs the completeness check numerically if all rules cover disjoint cases; otherwise, the check is done by exhaustive enumeration. Numerical completeness check considers the actual cases that a rule covers.

2.3.3.2 Petri net-based definitions

INDE [Pipard, 1988] is a Petri net-based system to check the consistency and completeness of KBs in the MORSE framework. Starting from a Petri net representation of the KB, the maximal sets of rules that fire simultaneously are obtained. Each set, more precisely its corresponding net, is checked for inconsistency by studying the attributes that receive two different values.

In [Meseguer, 1990] propositional logic KBs are transformed into Petri nets in order to translate the question about consistency into a reachability problem in nets. It is shown that this problem is equivalent to solving a linear equation system.

2.3.4 Logical analysis

The technique in [Ligeza, 1997] is devoted to the logical analysis of completeness of single-layered rule-bases, i.e. rule bases where rule consequents are actions and no chaining takes place. Completeness analysis is based on the use of backward dual resolution (bd-resolution) which is an inference method dual to classical resolution. For example, given two propositional logic formulas $\psi_1 \wedge \omega$ and $\psi_2 \wedge \neg\omega$, where ω is a propositional symbol, the basic form of bd-resolution is:

$$\frac{\psi_1 \wedge \omega,\ \psi_2 \wedge \neg\omega}{\psi_1 \wedge \psi_2}$$

which means that the disjunction of the parent formulas is the consequence of their bd-resolvent $\psi_1 \wedge \psi_2$. When it is applied to the premises of two rules, bd-resolution shows that they constitute a complete set of rules for any situation satisfying the bd-resolvent. Indeed in any such situation either ω or $\neg\omega$ must hold and therefore one of the rules can be applied.

2.3.5 Summary

The previous techniques are intended for the verification by detection of implementation-dependent anomalies in the rule-based knowledge bases. Some important characteristics include the following:

- In general their final objective is providing automatic or semi-automatic methods.
- The anomalies that they consider are mainly inconsistency and incompleteness. The techniques to check for incompleteness, with the exception of the work in [Ligeza, 1997], are based on exhaustive enumeration of possible combinations of inputs.
- The rule-based representation formalisms that they assume are propositional logic or slightly extended languages (e.g. AOV rules).
- The deduction model that they assume is monotonic and non-selective, i.e. without any conflict resolution strategy. Regarding this point, some authors surprisingly emphasise that KB anomalies must be defined independently of the inference engine that will use it [Ayel, 1988], [Mellis and Ruckert, 1989]. Many others, however, are aware of the influence of the deduction model characteristics.

The table in figure 2.1 summarises the presented techniques. Some work focuses on the improvement of the computational complexity of the verification process, particularly:

- **Grouping of checks.** In the COVER KB verification tool [Preece and Shinghal, 1992], [Preece et al., 1992] checks for anomalies are grouped into different procedures according to their theoretical computational complexity. For instance, checks based on pairwise rule comparisons (named rule checks) and those considering all possible deductions (rule extension checks) are performed separately. The idea is performing first the unexpensive checks, which often indicate errors with a high likelihood.

Approach	**System/Technique**	**Anomalies**	**Knowledge representation**
Rule syntax-based definitions	ONCOCIN rule checker [Suwa et al., 1982]	inconsistent, redundant, subsumed and missing rules → (a)	ONCOCIN rules
	CHECK [Nguyen et al., 1985]	(a) + circular rule chains, dead-end and unreachable clauses → (b)	LES rules
	ARC [Nguyen, 1987]	(b) + unnecessary conditions, redundant, subsumed and conflicting rule chains	ART rules
Deduction-based definitions	GCE [Beauvieux and Dague, 1988]	inconsistency	propositional logic and AOV-like rules
	COVADIS [Rousset, 1988]	inconsistency	MORSE rules
	KB-Reducer [Ginsberg, 1988]	inconsistency and redundancy	propositional logic
	COCO [Loiseau, 1992]	inconsistency	first order logic
	[Meseguer, 1992]	redundancy, inconsistency, circularity, and useless objects	modular rule bases
Translation-based definitions	[Puuronen, 1987]	inconsistency, redundancy and incompleteness	ONCOCIN-like rules
	ESC [Cragun and Steudel, 1987]	redundancy and incompleteness	AV-like rules
	INDE [Pipard, 1988]	inconsistency and incompleteness	MORSE rules
	[Meseguer, 1990]	inconsistency	propositional logic
Logical definitions	[Ligeza, 1997]	incompleteness	first order logic, single-layered rule bases

Figure 2.1: *Techniques for the verification of knowledge-based systems by KB anomaly detection*

- **Incremental nature of checks.** Many checkers, after KB modification repeat checks on the whole KB ignoring the previous verification results. The basic idea underlying incremental verification is to repeat only the tests on those parts for which previous results are not guaranteed to hold. As examples we can cite [Beauvieux and Dague, 1988], [Meseguer, 1992], and [Loiseau, 1992].

2.4 Evolution of the field

An evolution can be observed in the field of V&V of knowledge-based systems [Nazareth and Kennedy, 1993]. First checkers, like ONCOCIN rule checker, aim at locating errors in particular knowledge-based systems by checking their KBs for anomalies (stand-alone checkers). With the maturity of development environments, next checkers target knowledge-based systems built in specific environments (environment-oriented checkers) or with specific knowledge representation formalisms (representation-oriented checkers). Most of the checkers that we have reviewed belong to these approaches.

Later work concentrates on tools to deal with different V&V problems, like e.g. consistency checks and test case generation. These tools are applicable to knowledge-based systems built with different environments, and are integrated into a single development verification environment. Examples are the EVA [Chang et al., 1990] and VALID [Meseguer and Plaza, 1992] verification environments. Both EVA and VALID environments follow a meta-level approach, which means that the tools work on a representation of the knowledge-based system. For instance, this representation includes in VALID structural and behavioural aspects of the knowledge-based system, but also objects needed for V&V purposes like KB versions or traces.

2.5 Discussion

Numerous V&V techniques involve verification by detection of KB anomalies defined in terms of the representation formalism of the KB, mainly rule-based. Typical anomalies are inconsistency and incompleteness. This approach has several drawbacks.

Firstly, implementation-dependent anomalies, though a prerequisite for the adequate functioning of the knowledge-based system, are not sufficient because they say little about the actual task that the system is required to perform [Meseguer and Preece, 1995]. For instance,

a consistent and complete rule base may perform badly because the task requires other properties that have nothing to do with rule base consistency and completeness.

Secondly, the verification by detection of such anomalies addresses knowledge-based system implementation. This approach dismisses the V&V activities that could take place earlier in the development. An exception to V&V activities addressing the implementation is the following recent work. In [Fensel et al., 1996], [Fensel and Schönegge, 1997] the V&V is based on a formal specification of the knowledge-based system task and is carried out by using a tool for software verification. The work in [Cornelissen et al., 1997] addresses also the V&V based on formal specifications. Here the characteristics of the specification framework, which decomposes a task into subtasks and so on, serves to structure the V&V process. A different approach is the work in [van Harmelen and Ten Teije, 1997] which exploits a conceptual model of the knowledge-based system task for V&V.

Third, the work on verification by anomaly detection concentrates on rule-based formal definitions for which automatic or semi-automatic methods can be implemented. This has been fostered by the close connection between rule-based representation formalisms and logics. Techniques to deal with systems implemented using other representation formalisms, e.g. hybrid representations including objects, have been disregarded. An exception to this is the work on the redefinition of rule subsumption anomalies in hybrid KBs in [Sunro and O'Keefe, 1993], which is still in the line of previous formal anomaly definitions.

The theoretical definitions of anomalies and their associated verification methods have been extensively studied. However, techniques or methodologies to approach the V&V of "real-world" knowledge-based systems have been disregarded, e.g. hybrid knowledge-based systems which perform well-understood tasks like planning.

2.6 Conclusions

The main drawbacks of the use of implementation-dependent anomalies for V&V are their lack of significance in relation to the actual task of the system and their focus on implementation, which delays V&V activities until late development phases.

To overcome the problem of the lack of significance of implementation-dependent anomalies, we must employ more precise information about the application domain and the task that the knowledge-based system is required to perform [Meseguer and Preece, 1995]. This implies

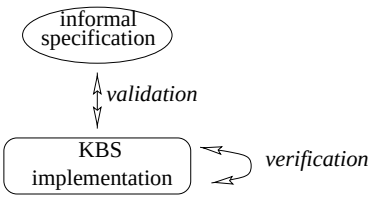


Figure 2.2: *V&V without any formal specification*

exploiting a conceptual model of the knowledge-based system which permits the definition of more significant V&V properties. To some degree this approach has already been advocated in [Ayel, 1988], which proposes the utilisation of a model of the KB for the definition of the consistency issues. Only recently this idea has started to win acceptance in the V&V field. Conceptual models also allow to perform some V&V before the implementation phase since they can act as informal or semi-formal specifications of the knowledge-based system.

To perform V&V before the implementation, the development process should include some kind of specifications. When only informal descriptions are available as specifications, V&V is limited to the verification of the internal consistency of the implementation and of its approximate compliance with the informal specifications as figure 2.2 shows. The introduction of one or more levels of formal specifications provides support for additional V&V activities [Meseguer and Preece, 1995]. For instance, in the situation of figure 2.3 we find formal specifications with different levels of detail. Validation would consist of the validation of the highest-level specification against the informal specification plus the verification of the correspondences between the rest of the levels.

Finally, we believe that more attention has to be paid to techniques or methodologies that support V&V even if not in an automatic way. In our view, only in this way we can aim at the V&V of knowledge-based systems with the characteristics of PS systems.

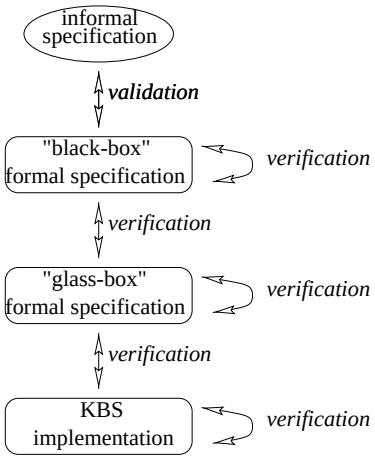


Figure 2.3: *V&V with two levels of formal specifications*

Chapter 3

Research Framework

Different aspects constitute the framework for the design and implementation of PS systems in our team. The most important ones are the conceptual model of PS and the Lama development platform. These aspects, especially the PS model, have influenced our approach to the verification and validation of PS systems.

In this chapter we first describe the activities involved in the PS task according to our view and the architecture of our PS systems. Then we detail the conceptual model of PS, with examples of how it can be exploited for the verification and validation of PS knowledge bases, and the development platform Lama. We finish by summarising our focus for the verification and validation of PS systems.

3.1 Program supervision task

PS techniques arise to give support to unexperienced users in the utilisation of sophisticated program libraries. The aim of the PS task is the automation of the activities involved in the skilled use of a program library.

Our view of the PS task is the following. To solve a user's request consisting of an intended processing goal and a set of data to operate on, the adequate programs must be selected, scheduled and executed, and the execution results must be monitored in order to ensure that the previously taken decisions were appropriate.

The PS task implies different activities or subtasks:

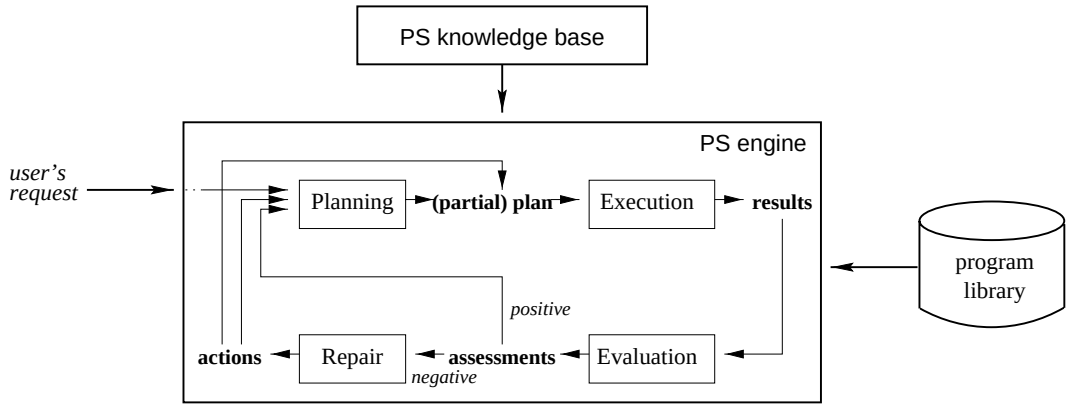


Figure 3.1: *Architecture of a PS system and reasoning strategy of the PS engine*

- The selection of a set of programs that can solve the user's request, and the actual scheduling of programs.
- The initial setting of program parameters, and the execution of programs.
- The evaluation of the results of program execution to ensure that the programs and/or their tuning were appropriate.
- The correction of scheduled programs, or the adjustment of program parameters, if the results are deemed unacceptable.

These activities demand a great deal of expertise on program utilisation. For instance, the expertise for program selection may include the input and output arguments of the program, the processing function that it performs, the situations in which it can be applied and/or its side-effects. Program scheduling may be based on the previous expertise or may use knowledge about typical combinations of programs.

3.2 Architecture of program supervision systems

PS systems embody the expertise involved in the skilled use of a program library in a knowledge-based architecture. In our case, a PS system is composed of a PS inference engine, a PS knowledge base encapsulating the expert's knowledge on the utilisation of the programs, and the library itself (see figure 3.1). The PS knowledge base includes e.g. the expertise for program selection described before.

In general, the PS knowledge base constitutes the domain-specific part of the PS system and the PS engine is the domain-independent one. This is not always the case since the knowledge base may include more or less generic knowledge, e.g. about the characteristics of a certain type of image, and the engine may partly reflect the reasoning process of the expert when solving a processing problem for a particular application. Despite this, the PS engine is in general intended to be reused across different applications within a more or less wide application domain.

V&V issues In this thesis we focus on the verification of PS knowledge bases intended for new applications. On the other hand, the V&V of PS engines is also considered since they can be adapted to better suit the reasoning strategy of the expert.

The previous concerns are in contrast with the usual view in the field of V&V of knowledge-based system, where the verification of the knowledge base has traditionally been considered the crucial issue for ensuring system reliability whereas the V&V of the inference engine is usually assumed.

3.3 Conceptual model of program supervision

Many different PS applications have been developed in our team, mainly from the OCAPI engine. As examples we can cite applications for road scene image processing [Thonnat et al., 1994], astrophysical image processing [Thonnat et al., 1995], and medical image processing [Crubézy et al., 1997]. These experiences have been the basis for the identification of the knowledge concepts and the reasoning strategy most commonly used in PS according to our view. These constitute a generic conceptual model of PS on which our current implementations are grounded, that is, both knowledge bases and inference engines more or less correspond to the patterns defined by the generic concepts and reasoning strategy. They are generic in the sense that they are common to most of our PS systems, although with variations in details.

V&V issues In the field of V&V of knowledge-based systems, a widely recognized approach consists in checking the knowledge base for a set of implementation-dependent anomalies such as inconsistency and incompleteness. As it has been highlighted in chapter 2, this approach is insufficient because such anomalies say little about the task that the system is intended to

perform. Nevertheless this task is well-understood in our case. Our claim is that with the help of our information about the task we can examine the knowledge base to determine its adequacy to the performance of the PS task.

In this thesis we exploit the conceptual model of PS for the definition of the properties which a knowledge base must be checked against. On the one hand, the knowledge concepts as well as their interrelations provide a good characterisation of the organisation in the knowledge base. On the other hand, the reasoning strategy of the inference engine provides information on the knowledge utilisation that can help determine the properties that the knowledge base should verify in order to adequately serve to perform the PS activities.

Next we present some details of the generic knowledge concepts and reasoning strategy, together with examples on how they can help us with the definition of the properties for the verification of the knowledge base. It must be noticed that this is only a preliminary sketch and that the full conceptual model will be presented later on.

3.3.1 Program supervision concepts

Many different knowledge concepts are used to perform the PS task. The main ones are *operators*, corresponding either to programs or typical combinations of programs, input and output *arguments* of the operators, and operator *parameters*, or tunable arguments. In PS systems where a high degree of automation is required, several expert *criteria* permit the system to perform automatically different actions such as the initial setting of operator parameters (*initialisation criteria*), the evaluation of the results of program execution (*evaluation criteria*) or the application of a corrective action in case of negative evaluation (*repair criteria*).

The operator concept is fundamental. An operator describes either an individual program or a more or less complex combination of programs. According to this there exist two types of operators, namely primitive and compound ones. Primitive operators describe individual programs. Compound operators constitute combinations of other operators, which can be in their turn primitive or compound ones. Operator combinations can denote e.g. a sequence. The execution of a sequential compound operator involves the ordered execution of the operators in the sequence.

In the following we find some examples of how PS knowledge concepts and their interrelations can help with the definition of properties useful for the verification of knowledge bases. Operators have at least one input argument and one output argument, and they may have

or may not have any parameter. A simple property that the knowledge base should verify is that operators with parameters have knowledge to initialise them, e.g. initialisation criteria.

Compound operators can be described at different levels of abstraction since the operators in a sequence e.g. can be themselves primitive or compound ones. At some point this description has to end with primitive operators; otherwise, the compound operator would not constitute a combination of programs. This is an important property that the compound operators in the knowledge base should verify.

Finally, it is important to note that the previous knowledge concepts are common to all our PS systems, although with variations in details. For instance, a rich description of operators may additionally include expertise about the processing function that they perform, and their characteristics, applicability conditions and side-effects.

3.3.2 Program supervision reasoning

The reasoning strategy of a PS engine can be roughly divided into four steps as figure 3.1 shows. First an initial *planning* step determines the best (partial) plan to reach the goals defined by the user's request. Then the *execution* of the (partial) plan is triggered, i.e. the individual programs in the plan are executed. Afterwards the results of program execution are passed on to an *evaluation* step that assesses their quality. This evaluation can be done either automatically by using the evaluation criteria in the knowledge base, or interactively by the user. Finally if the assessment on results is negative, a *repair* step decides the appropriate corrective action making use of the repair criteria in the knowledge base. Otherwise the process continues with the planning step for the remaining (sub)goals, if there are any.

As an example of how the PS reasoning strategy can be of help for the verification of the knowledge base we can cite the property concerning the mutual dependence of the expertise employed in the evaluation and repair steps, i.e. the evaluation and repair criteria. Actually, as the repair step is performed if the evaluation one concludes a negative assessment on the execution results, some repair criteria must exist whenever the evaluation criteria may conclude a negative assessment.

Notice that, although the described behaviour is quite general, variations are possible at different levels. At a high level, for instance, planning and execution can be interleaved because the planning step may depend on information only available after the execution of previous programs in the plan. This is the case in OCAPI engine, which complies with the

generic reasoning strategy that we have described. At a lower level, basic steps within the mentioned ones can be performed in a more or less complex way. For instance, the planning step makes use of a step for operator selection for which different alternatives exist.

3.4 LAMA platform for the development of program supervision systems

The PS systems that are currently developed in our team are implemented under the LAMA platform. LAMA is a software platform for the development of knowledge-based systems which is at present devoted to the construction of PS systems, both to inference engine and knowledge base design [Crubézy et al., 1998]. The rationale of LAMA is to facilitate inference engine (re)configuration. This objective is sought by means of reusable, PS task-oriented components for the configuration of PS inference engines. LAMA shares ideas with second generation knowledge-based systems [David et al., 1993].

The primary LAMA constituent is the BLOCKS library of components for inference engine and knowledge base design. BLOCKS components are implemented on top of Le-Lisp [Le-Lisp, 1991]. A more efficient and portable version of the BLOCKS library, based on C++, also exists. The BLOCKS library provides two types of components, namely data structures and instructions [Vincent et al., 1996]. Some of the components are general-purpose ones (e.g. stack or pop from a stack) whereas others are based on the conceptual model of PS, or PS model-based (e.g. operator or execute-operator). Model-based components have been inspired by the above conceptual model: model-based data structures correspond with PS knowledge concepts, and model-based instructions are a set of small-grain sized functions intended for the configuration of PS inference engines.

All BLOCKS components are free of implementation details. Indeed model-based structures can be implemented using alternative representation formalisms. What is more important, BLOCKS model-based components lie at a level of abstraction which is close to PS. Particularly, model-based instructions are closer to PS reasoning than ordinary programming languages, in order to help engine designers in the configuration of new engines and the adaptation of existing ones.

Another important element of LAMA is the YAKL language for expertise description. The YAKL language supplies a set of syntactic constructs allowing to express the contents of PS knowledge bases at a higher level of abstraction than BLOCKS PS structures.

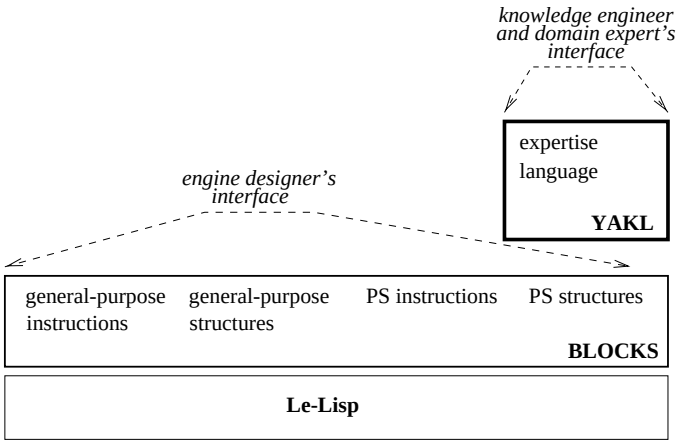


Figure 3.2: *The different layers of the* LAMA *platform.* The layer of BLOCKS components is the interface used for the configuration of inference engines. The YAKL language layer is the interface used for knowledge base development. Each layer depends on the one immediately below: BLOCKS components are implemented on top of Le-Lisp and YAKL constructs are high-level counterparts of the BLOCKS PS structures.

LAMA provides two different interfaces for the configuration of inference engines and the development of knowledge bases, namely the BLOCKS library and the YAKL language. Figure 3.2 shows these two interfaces. To build a PS inference engine, the designer combines BLOCKS instructions and structures in an algorithmic way. For knowledge base development, the knowledge engineer and domain expert exclusively use the YAKL constructs. These constructs are afterwards translated into the corresponding BLOCKS PS structures.

In addition to BLOCKS and YAKL layers, the LAMA platform comprises different modules. Figure 3.3 shows the modules involved in the utilisation of a PS system by the end-user, namely the communication module and the graphical user interface for the visualisation of knowledge bases and the execution of PS sessions.

V&V issues The V&V issues we focus on are adequate to the needs of the LAMA platform. First, both the V&V of inference engines and knowledge bases are essential in the development of PS systems that LAMA fully supports. And second, the use of properties based on the PS conceptual model for knowledge base verification is in accordance with the model-based nature of the YAKL language. Domain experts, who use the model-based YAKL constructs to describe their expertise, will find such properties much more significant than implementation-dependent ones.

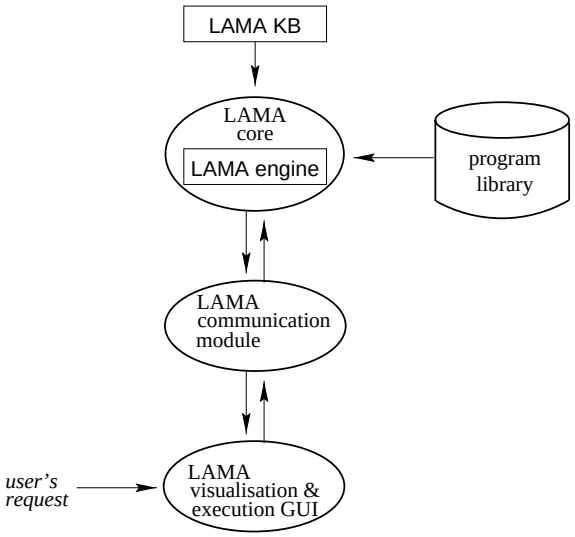


Figure 3.3: *The utilisation of a PS system in the* LAMA *platform.* The end-user interacts with the LAMA PS system through the LAMA graphical user interface (GUI). The LAMA communication module is in charge of the communication between the GUI and the LAMA PS system.

3.5 Conclusions

Summarising, our main focus in this thesis is the verification of PS knowledge bases. The verification and validation of PS inference engines is also considered since they can be adapted to meet the reasoning strategy of the expert.

For the verification of PS knowledge bases, we have shown that the conceptual model of PS can be helpful in the definition of certain properties that the knowledge base should verify. These model-based properties are defined from the knowledge concepts and their interrelations, and the information about their utilisation during PS reasoning. Such properties, especially those derived from knowledge utilisation, refer to the task that the system is intended to perform and hence they are much more significant than the implementation-dependent ones traditionally used in the field of V&V of knowledge-based systems.

Finally, both the model-based verification of PS knowledge bases and the V&V of PS inference engines are adequate to the development process of PS systems with LAMA. In chapter 4 we present the approach that we have adopted to put these ideas into practice.

Chapter 4

Approach to the Verification and Validation of Program Supervision Systems

The research framework that we have described in chapter 3, especially the conceptual model of PS, has influenced our focus for the verification and validation of PS systems, namely the verification of knowledge bases using information on knowledge organisation and utilisation, and the V&V of inference engines.

In this chapter we present how we approach the V&V of knowledge bases and inference engines, and describe the work directions that have been undertaken accordingly, which correspond to the next three chapters. We finish with a comparison of our positioning with similar work in the knowledge engineering area.

4.1 Verification and validation of knowledge bases

The verification of knowledge bases is crucial to ensure the reliability of PS systems. The techniques presented in chapter 2 provide us with a variety of definitions for the verification of rule-based knowledge bases. These techniques do not suit our needs as they are for two reasons. First, since they are based on the implementation they cannot capture the model-based issues such as the interdependence of evaluation and repair knowledge that we mentioned

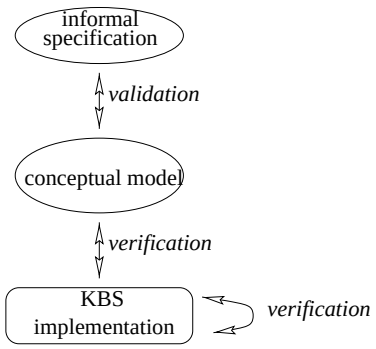


Figure 4.1: *Model-based verification*

in chapter 3. Second, they exclusively deal with rule-based representations which, though fundamental in PS, are not the sole representation formalisms used in LAMA.

To overcome the limitations of the use of these techniques we propose firstly to exploit the knowledge **organisation** and the knowledge **utilisation** during reasoning in order to define interesting properties for the verification, similarly to the examples presented in chapter 3. Next we propose to apply the implementation-dependent verification techniques that are appropriate according to the target **representation** formalism. Knowledge utilisation especially determines the required knowledge, together with the roles that the knowledge plays in reasoning [Marcos et al., 1995]. This can be exploited for the definition of properties that serve to determine the adequacy of the knowledge base to the reasoning of the inference engine.

In this way, given a knowledge base and the intended PS inference engine, our aim is not only the verification of implementation-dependent properties, e.g. non redundancy, but rather the verification of the adequacy of the embodied knowledge to the way in which it will be used by the inference engine. We use the term *model-based verification* to denote our approach [Marcos et al., 1997].

Our model-based approach to the verification of knowledge bases consists in the verification of properties defined at the conceptual model level, as it has been expressed in figure 4.1. Such properties refer to the PS task and use the terminology of the task, e.g. *"initialisation criteria x of operator y are adequate"* instead of *"rule base x of operator y is complete"*. Thanks to this, they are more interesting than the usual implementation-dependent properties, and much more significant for the domain expert. These properties can also be exploited prior to the implementation of the knowledge base.

With the purpose of understanding the knowledge utilisation by the PS inference engine, a thorough analysis, free of implementation concerns, of its problem-solving behaviour is required. A knowledge-level analysis[1] of PS problem-solving suits our needs well. The notion "knowledge-level" was introduced in [Newell, 1982] as a way to represent domain or problem-solving descriptions independently of how they are implemented (opposed to "symbol-level"). Knowledge-level descriptions are not only useful for enhancing human understanding but also for their system engineering benefits [Uschold, 1998].

A number of knowledge engineering frameworks deal with problem-solving descriptions at the knowledge-level. Next we detail the CommonKADS framework, which has been widely used by the knowledge engineering community, and describe another interesting one in the spirit of CommonKADS. In the light of the latter we reformulate the model-based verification of PS knowledge bases.

4.1.1 Knowledge-level frameworks for knowledge-based systems

Problem-solving descriptions at the knowledge-level are typically expressed in terms of problem types, tasks and problem-solving methods, and the relationships among these. The existing knowledge-level frameworks differ in the knowledge categories they use and in how they are structured.

4.1.1.1 CommonKADS framework

CommonKADS [Schreiber et al., 1994] is a particularly important effort addressing problem-solving descriptions. It provides a wide framework, encompassing all knowledge pertinent to the development of knowledge-based systems. The CommonKADS model set [de Hoog et al., 1994] provides models to capture the organizational context of the system (organisation model), the tasks supported and the distribution of tasks over different agents (task model), the capabilities of agents (agent model), the communication between agents (communication model), and the computational system design (design model). A central model is the expertise model, which describes problem-solving behaviour of an agent in terms of the knowledge that is applied to perform a task. It constitutes a problem-solving description in a narrower sense.

[1] The terms knowledge-level analysis, knowledge-level modeling and knowledge modeling are indistinctively used in this thesis.

In the CommonKADS expertise model [Schreiber et al., 1994], [Wielinga et al., 1994], the knowledge categories to describe a knowledge-based system are as follows:

- *domain knowledge*, which is a description of the knowledge relevant to the application, independent of the role that it plays in reasoning.
- *inference knowledge*, that describes the basic reasoning steps that can be performed using domain knowledge together with the roles that it plays, and the possible ways in which inference steps can be combined.
- *task knowledge*, which describes the decomposition of the top-level reasoning task of the system, and the control in this decomposition.
- *problem-solving knowledge*, including problem-solving methods which are descriptions prescribing how a task can be achieved, including the main steps and the control over them.

These knowledge categories are described independently to facilitate reuse, e.g. the reuse of a problem-solving method for solving different tasks.

4.1.1.2 A CommonKADS-based framework

Different frameworks regard relationships between tasks and problem-solving methods, like the CommonKADS expertise model. A common view is that a problem-solving method can be applied to a task provided that some conditions are satisfied [Benjamins and Pierret-Golbreich, 1996]. These applicability conditions refer to problem-solving method features and can be anything at all, including domain knowledge requirements. The latter make the connection between the problem-solving method and the domain knowledge explicit. They have been successfully exploited for knowledge engineering purposes, e.g. to find out to which domains a certain problem-solving method is applicable, or which problem-solving method is more suitable for a given domain [Benjamins et al., 1996b].

Applicability conditions have been designated in several ways, e.g. method ontologies [Gennari et al., 1994], suitability criteria [Benjamins, 1995], and assumptions [Benjamins and Pierret-Golbreich, 1996], [Benjamins et al., 1996a], [Fensel et al., 1996].

In [Fensel et al., 1996] we find a framework for the specification of knowledge-based systems that comprises different types of assumptions. It has been developed in accordance with the CommonKADS expertise model. In this framework three reusable elements are distinguished in the specification of a knowledge-based system:

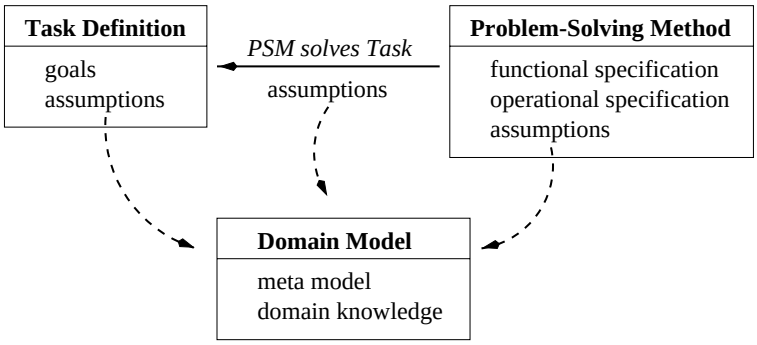


Figure 4.2: *The elements in the specification of a knowledge-based system and their assumptions*

- *task definition*, which describes the problem that the system solves.
- *problem-solving method*, that describes the reasoning process of the system. It includes a functional specification defining the competence of the problem-solving method independently of its realisation, and an operational specification of how this competence is achieved, i.e. reasoning steps and control over them.
- *domain model*, describing the domain knowledge structured as the task and problem-solving method require for reasoning. It includes the domain knowledge and a characterisation of its properties, e.g. its organisation.

Different types of assumptions relate the elements of a knowledge-based system (see figure 4.2). On the one hand, the task definition imposes certain assumptions on domain knowledge. On the other, the problem-solving method makes its own assumptions on domain knowledge. Finally, additional assumptions are often needed to ensure that the problem-solving method is able to solve the task. These assumptions either strengthen the problem-solving method or weaken the task.

4.1.2 Model-based verification of knowledge bases

In the light of the specification framework above, our approach to the verification of knowledge bases is based on the organisation in the domain model and on the assumptions that the PS task and, above all, the PS problem-solving method make on the domain knowledge (i.e. knowledge base) for a particular application.

Some examples of properties defined from these elements follow. The organisation in the domain model, which corresponds to an elaboration of the generic knowledge concepts

presented in chapter 3, imposes several structural properties, e.g. operators must have at least one input argument and one output argument. The PS task necessitates e.g. knowledge about the use of a set of programs. The operator selection step in the PS problem-solving method may use a heuristic based on operator side-effects and thus require that this knowledge be available. Since the assumptions of the task are rather general we will concentrate on the ones dealing with the problem-solving method.

A PS inference engine implements a particular PS problem-solving method (PS method henceforth) which makes use of a PS domain model. First, the knowledge base must conform to the knowledge organisation in the PS domain model. Second, the operational specification of the PS method imposes additional assumptions on the knowledge base, namely the required knowledge and possibly the characteristics that it must fulfil so that the method can perform properly. These elements constitute the definitions for knowledge base verification that we are searching for.

According to this, we have carried out a knowledge-level analysis of different PS engines that are currently used in our team. This analysis comprises the description of both the knowledge organisation in the domain model and the knowledge utilisation by the PS methods. From this knowledge model we have identified the assumptions that PS methods make on domain knowledge. These assumptions have been exploited in the design and implementation of a module for the verification of PS knowledge bases. The knowledge modeling of PS systems is presented in chapter 5. The verification module that has been developed accordingly is presented in chapter 6.

4.2 Verification and validation of inference engines

The V&V of PS engines is very important since they can be modified according to the particular reasoning strategy of the domain expert. PS engines are software and as such they can be verified using formal techniques for traditional software.

Formal verification is a difficult job for which the support of an adequate tool is fundamental. The KIV software verification tool has been applied to the verification of knowledge-based systems using simple problem-solving methods [Fensel et al., 1996], [Fensel and Schönegge, 1997]. Precisely, KIV is used for the verification of the adequacy of a problem-solving method to solve a task. An interesting aspect of KIV is that it is also used to guide the detection process of the assumptions necessary to ensure that the problem-solving method can solve the task.

Inspired by the results obtained in the previous work, we have carried out some experiments to assess the feasibility of the utilisation of KIV for the V&V of inference engines and the detection of the assumptions that their PS methods impose on domain knowledge. Our experiments consist in the specification and verification of one of the reasoning steps that have been identified in the knowledge model of chapter 5. In chapter 7 we report on these experiences.

4.3 Discussion

For the V&V of knowledge bases, the field of knowledge-based systems has attained maturity regarding the variety and quality of implementation-dependent techniques available at present. We propose applying these techniques in combination with an idea that has already proven to be useful in the knowledge acquisition area, namely the benefits of understanding knowledge utilisation.

The interest in understanding knowledge utilisation in problem-solving is not new in the knowledge acquisition area. Role-limiting methods [Marcus, 1988] and PROTÉGÉ [Musen, 1989] are examples of special-purpose knowledge-based systems, each one with its particular knowledge acquisition tool. Special-purpose problem-solvers present the advantage of clarifying how knowledge is used, thus providing a set of expectations that can guide the knowledge acquisition process [Musen, 1992], [David et al., 1993]. We share with role-limiting methods and PROTÉGÉ the idea that a knowledge base can be examined to judge its adequacy to perform the task for which it is intended. This can be exploited not only for knowledge acquisition but also for the V&V of knowledge correctness and completeness [Uschold, 1998].

The V&V of inference engines has been disregarded in the field of knowledge-based systems because it is considered to be a software engineering concern. However, recent work has demonstrated the utility of software verification techniques to detect the domain knowledge assumptions that are necessary to ensure that a problem-solving method can be applied to solve a task. This research stems from the knowledge engineering field and is motivated by the growing interest in developing knowledge-based systems from reusable components, e.g. libraries of problem-solving methods [Fensel et al., 1996], [Fensel and Groenboom, 1997]. We propose the verification of PS methods with the purpose of not only ensuring that they are reliable and can solve the PS task, but also as a formal means for the detection of the

Chapter 5

Knowledge Modeling of Program Supervision Systems

The verification of knowledge bases is crucial to ensure the reliability of PS systems. The model-based approach that we have chosen presupposes a thorough understanding of both the knowledge organisation that PS engines require and their problem-solving behaviour, which determines their knowledge utilisation. With this purpose, a knowledge-level analysis of different PS engines has been carried out. We have analysed three PS engines that have been implemented in our team: PEGASE, PULSAR, and MedIA.

This chapter is structured as follows. First we detail the elements of the CommonKADS expertise model which we have used for knowledge modeling. Afterwards we present the three PS engines and their knowledge model. We finish with an enumeration of the benefits that we have gained from it and with a comparison of our analysis with similar modeling work.

5.1 CommonKADS expertise model

We have used different elements from the CommonKADS expertise model in our analysis. In CommonKADS, the primary knowledge categories to describe an application are domain knowledge, inference knowledge and task knowledge.

The *domain knowledge* contains a description of the domain knowledge relevant to the application. It comprises the entities (concepts) and relationships between entities (relations) needed to reason about the application.

The *inference knowledge* describes the basic inferences (inference steps) that can be made using the domain knowledge, and the roles that it plays in these inferences (knowledge roles). Inference structures show the way in which inference steps can be combined through knowledge roles, without defining the flow of control.

The *task knowledge* describes how to decompose the top-level reasoning task of the system, and how to impose control in this decomposition. A task is characterised by two parts: the task definition, which is a declarative specification of the task goal, and the task body, which is a procedural program describing the activities to accomplish the task. The nature of the task body allows us to distinguish three types of tasks: tasks that are further decomposed into subtasks (composite tasks), tasks related to inferences (primitive tasks), and tasks of interaction with the world (transfer tasks).

Tasks (and subtasks) can be represented as inference structures in which the inference steps corresponding to composite tasks are further decomposed into new inference structures. Besides, a hierarchical decomposition of the task (or task decomposition) can be used to provide a global view of task knowledge.

5.2 Program supervision engines: PEGASE, PULSAR and MedIA

The knowledge modeling that follows describes three PS engines implemented in our team: PEGASE, PULSAR, and MedIA. All of them are successors of the former OCAPI PS engine and improve it in different directions. The three engines have been developed under the LAMA platform as explained in [Crubézy et al., 1998].

OCAPI [Thonnat et al., 1994] works on operators described at different levels of abstraction. It performs a selection of operators to solve the initial problem and then searches for a solution in the set of selected operators, that is, an operator that actually solves the problem after hierarchical refinement. OCAPI thus combines operator-based planning (STRIPS-like [Russel and Norvig, 1995]) and hierarchical planning. The latter performs an interleaved planning and execution, like the generic strategy described in chapter 3, which is typically convenient for the supervision of image processing programs. OCAPI provides the re-execution of primitive operators as the only repair mechanism.

A knowledge-level analysis of OCAPI and the different experiences from its utilisation have served to identify some deficiencies [van den Elst, 1996]. In particular, the insufficient repair

capabilities and the lack of flexibility of the planning strategy has motivated the improvements introduced by PEGASE, PULSAR and MedIA.

PEGASE [Vincent, 1997] incorporates richer repair mechanisms and some innovations for sequential combinations of operators, namely the possibility of processing optional operators on the basis of some expert criteria.

PULSAR (in concrete PULSAR-IU in [van den Elst, 1996]) tries to solve the lack of flexibility of OCAPI. It also combines operator-based and hierarchical planning but in a more flexible way. It performs an operator-based planning step, then a hierarchical planning step, and, if a solution has not been obtained after this, it continues in a loop with the operator-based and hierarchical planning steps. PULSAR uses a subset of PEGASE repair mechanisms.

MedIA [Crubézy, 1999] also uses operator-based and hierarchical planning, improving the lack of flexibility of OCAPI strategy in a different manner. It incorporates abstract steps within sequential combinations of operators, representing subproblems to be solved at run-time. MedIA uses operator-based planning both to solve the original problem and the subproblems arising from abstract steps.

5.3 Knowledge modeling of PEGASE, PULSAR and MedIA

According to our objectives, the knowledge model of a PS engine should include descriptions of the organisation that it enforces on the domain model and of its problem-solving behaviour. For both purposes we have made use of part of the notions that the CommonKADS expertise model supplies: we have used concepts and relations to describe the organisation in the domain model, and inference structures and task definitions to describe the reasoning of the PS method. As modeling software we have employed Kadstool [Kadstool, 1993].

In the description of the underlying organisation of the domain model, a single version is presented which comprises the specification of the concepts and relations required by the three engines, together with an indication if they are specific to any of them. It has been referred to as PS domain model; henceforth, when we use this term we will refer to a (meta)model describing the organisation of the domain model as it is required by PS methods.

Next we present the PS domain model, the PS task and the different PS methods that PEGASE, PULSAR and MedIA implement, and their assumptions.

5.3.1 Program supervision domain model

Herein we present the PS knowledge concepts structured as PS methods require for reasoning. It must be noticed that we do not describe terms belonging to a particular application, such as `stereo-vision-matching` or `left-image`, rather, we describe the corresponding ones that PS methods use, i.e. operator or data.

The graphical notation used in Kadstool to describe domain knowledge follows the OMT [Rumbaugh et al., 1991] object-oriented notation. In OMT boxes represent concepts (classes) and contain the name of the concept. Lines between concepts represent relationships and are labelled with the name of the relationship. N-ary relationships are indicated with diamonds. Both concepts and relationships may have attributes. The multiplicity of a relationship, or number (minimal and maximal) of concept occurrences that may be associated to a single occurrence of the related class can be: one (line segments without any indication), one or more (indicated with 1+), two or more (with 2+), zero or one (with $\circ$), or zero or more (with $\bullet$). A concept (class) can be related to one or more refined versions of it (subclasses) constituting a hierarchy, which is represented with a triangle. Ovals indicate expressions making reference to any other element.

The knowledge concepts taking part in PS reasoning not only include the basic ones that have been presented in chapter 3 (*operator*, *argument*, etc.), but they also include the concepts necessary to describe the PS task, such as the *operator knowledge base*, and the *problem specification* and *problem solution*.

5.3.1.1 Basic concepts

Data Programs operate on data and therefore the data concept is fundamental in PS. All the important aspects of data should be represented. The following data subtypes leave the specification of what is important in every specific application open (see hierarchy in figure 5.1):

- simple data, to represent data with no additional structure. This concept has attributes to define an actual value (value), a default value (default), and a set of possible values (range). Different subtypes of simple data are predefined, such as integer or symbol.
- structured data, to describe data having different subparts, which is modeled through a relation between structured data and data (see relation in figure 5.1).

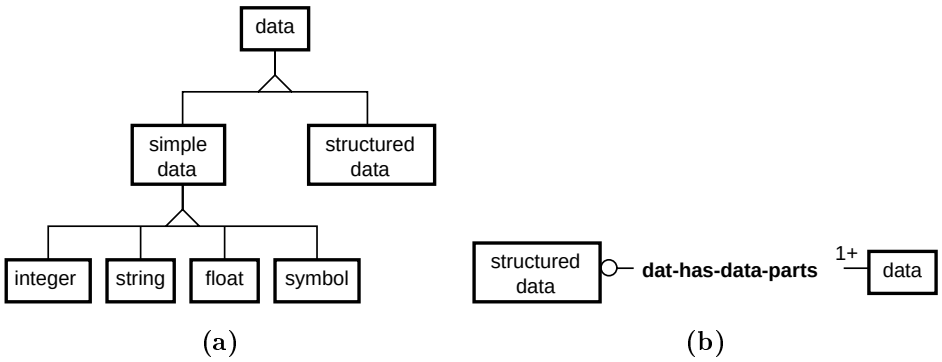


Figure 5.1: *Elements for data description:* **(a)** hierarchy of concepts to describe data types, and **(b)** relation to describe structured data

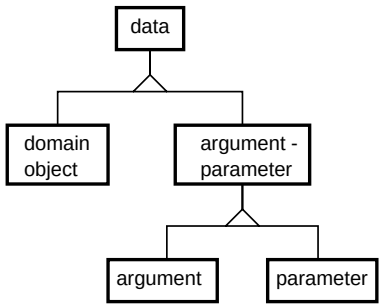


Figure 5.2: *Hierarchy of concepts to describe data utilisation*

The data subtypes argument, parameter and domain object serve to describe program arguments and parameters, and the objects constituting the execution context in PS (see hierarchy in figure 5.2). Arguments and parameters include the extra attribute assessment to hold the result of the PS evaluation subtask.

Different expressions on data are used along the PS domain model. The most frequent one is data reference, which represents a reference to any data attribute, including its value, e.g. an expression of the form $\langle data \rangle.\langle attribute \rangle$. Additionally, a data relation represents a comparison between data references, e.g. $\langle data\ reference \rangle\ \langle relation \rangle\ \langle data\ reference \rangle$.

In PULSAR possible dependences between data can be described from data references on the basis of the relation in figure 5.3.

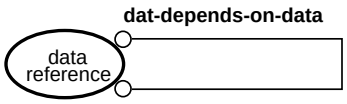


Figure 5.3: *Data dependences*

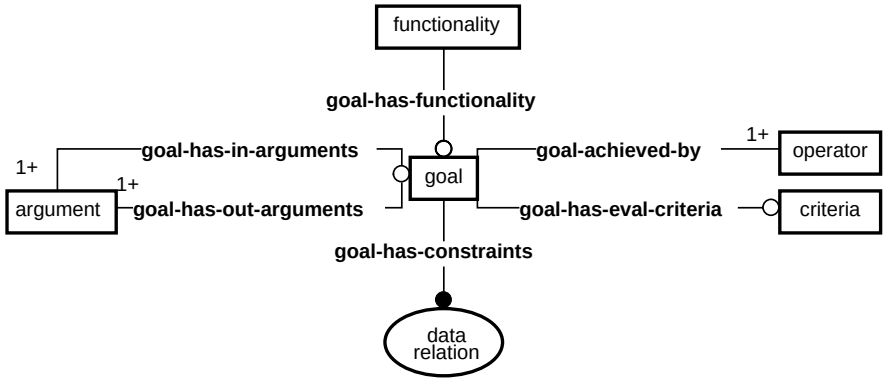


Figure 5.4: *Goal and related elements*

Goal A goal primarily represents a processing function that a PS system can perform (modeled through a relation with functionality), transforming a set of input arguments into the described output arguments (modeled, respectively, by the relations goal-has-in-arguments and goal-has-out-arguments). It serves to establish a link between a processing function and the operators that achieve it (relation goal-achieved-by). In this way goals permit the description of operators at a higher level of abstraction. Figure 5.4 shows these elements and other related ones.

Operator An operator describes either an individual library program, which we call primitive operator, or a more or less complex combination of programs, which we refer to as compound operator.

Within an operator we find knowledge about the function that it performs, to allow its selection in the appropriate situations, and about the resolution process that it employs [van den Elst, 1996]. Some knowledge is common whereas other is specific to either primitive or compound operators. We first describe the common knowledge and continue afterwards with the specifics of each type of operator.

Most of the common knowledge regards the function that the operator performs:

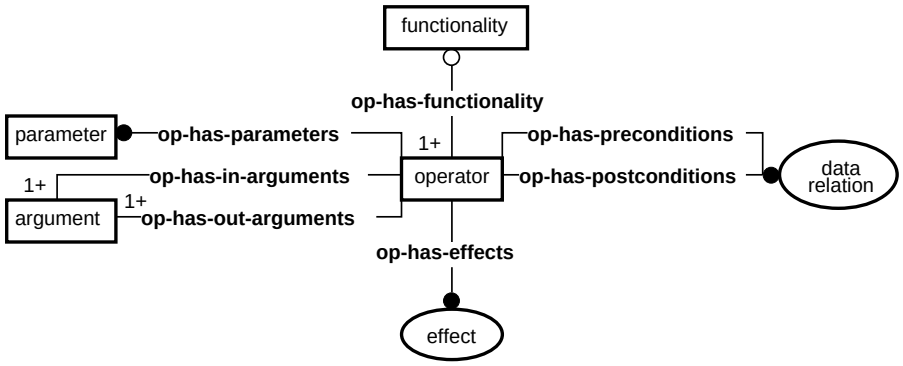


Figure 5.5: *Operator and related elements*

- processing function that the operator performs, which is modeled by the relation op-has-functionality.
- characteristics or typical properties of the operator.
- input and output arguments, modeled by the relations op-has-in-arguments and op-has-out-arguments, respectively.
- parameters or tunable arguments, modeled through the relation op-has-parameters.
- preconditions or conjunction of logical formulas stating the applicability conditions of the operator (relation op-has-preconditions). The formulas are expressions about the input arguments or the domain objects belonging to the execution context (data relations).
- different expressions representing the operator side-effects (relation op-has-effects). An effect has the form $\langle data\ reference \rangle := \langle data\ reference \rangle$.
- postconditions or conjunction of logical formulas that must hold after execution (relation op-has-postconditions). As in the case of preconditions, the formulas in postconditions can be seen as data relations.

Figure 5.5 pictures most of the elements described above. Additionally, an operator contains different expert criteria necessary to perform important PS subtasks (see figure 5.6):

- initialisation criteria, to initialise parameter values before execution (relation op-has-init-criteria).
- evaluation criteria, to check execution results and detect and diagnose potential problems (relation op-has-eval-criteria). The diagnosis is usually expressed as an assessment

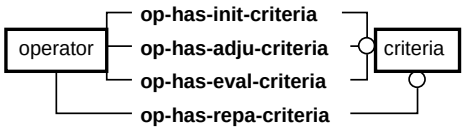


Figure 5.6: *Operator expert criteria*

on the operator application and/or tuning, combined with an assessment on an argument or parameter.

- repair criteria, to indicate how diagnosed problems have to be solved (relation op-has-repa-criteria). The alternatives are a local repair of the operator, which implies the adjustment of operator parameters and a further re-execution, or a non-local repair, which implies transmitting the problem to a previously executed operator considered responsible for the unacceptable results, or to a previous choice point. In case of operator re-execution, the adjustment criteria serve to adjust parameter values (relation op-has-adju-criteria).

Operators, in the same way as arguments and parameters, include an attribute assessment to keep the result of the evaluation subtask.

Primitive operator A primitive operator describes a library program, so it additionally includes information concerning the calling interface of the program or program call.

Compound operator A compound operator represents a combination of other operators, or decomposition. The operators in a decomposition can be in turn primitive or compound ones. By means of successive decompositions compound operators are described at different levels of abstraction, constituting a *hierarchical operator*.

The decomposition of a compound operator expresses how its function can be achieved using the suboperators in the decomposition. There exist different decomposition types to express the diverse manners in which suboperators can be combined. For instance, PUL-SAR admits unordered decompositions, which represent a set of suboperators to achieve the operator function without imposing any order over them. We focus on the most frequent decomposition types:

- sequential decomposition, expressing a sequence of suboperators that serves to perform the operator function.

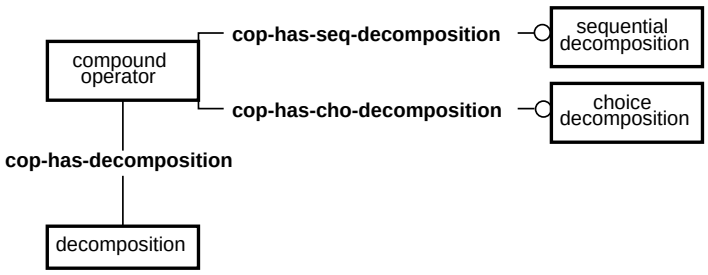


Figure 5.7: *Operator decomposition*

- specialisation (or choice) decomposition, corresponding to alternative suboperators to perform the operator function.

This is modeled by the concept decomposition and its subtypes (sequential decomposition and choice decomposition). A compound operator must necessarily be associated to a decomposition of one of these types describing how its function is achieved, as the relation cop-has-decomposition in figure 5.7 shows.

All decompositions are related to a suboperator set, but the complementary knowledge varies depending on the semantics of the decomposition. Figures 5.8 and 5.9 picture the relevant features of sequential and specialisation decompositions. They are related to two or more operators and in the case of sequential decompositions the order is relevant (see order attribute in figure 5.8). Choice decompositions need specific expert criteria, called choice criteria, allowing the selection of the candidates for the specialisation at execution time (relation cho-dec-has-choice-criteria in figure 5.9).

Engine specifics related to sequential decompositions are described next. PEGASE and MedIA permit optional suboperators, of which the application is determined at execution time on the basis of some expert criteria called applicability criteria. In addition to this, MedIA allows for the inclusion of abstract steps representing a functionality to solve and not only concrete suboperators. These specific characteristics, nevertheless, have not been included in figure 5.8.

To complement the knowledge on how its function is achieved, a compound operator must additionally specify:

- data distribution or the way in which input arguments or parameters are connected to the input of suboperators (and similarly, the way in which output arguments of suboperators are connected to the output of the operator).

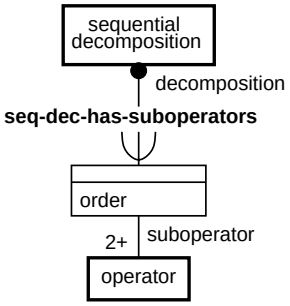


Figure 5.8: *Sequential decomposition*

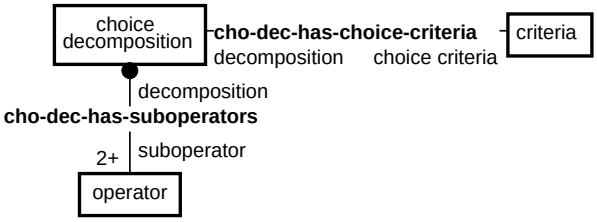


Figure 5.9: *Choice decomposition*

- in sequential decompositions, data flow or the connections among output and input arguments of the suboperators in the sequence.

The specification of the data distribution and flow within a compound operator can be seen as a set of expressions of the form $\langle operator_i \rangle.\langle argument_j \rangle \rightarrow \langle operator_k \rangle.\langle argument_l \rangle$. The relations cop-has-dat-distribution and cop-has-dat-flow in figure 5.10 model this kind of expressions.

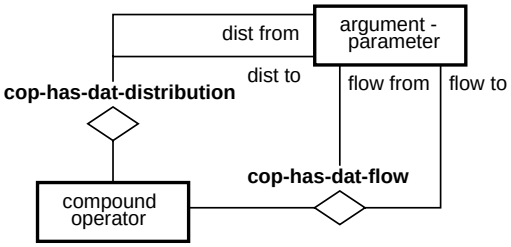


Figure 5.10: *Data distribution and data flow*

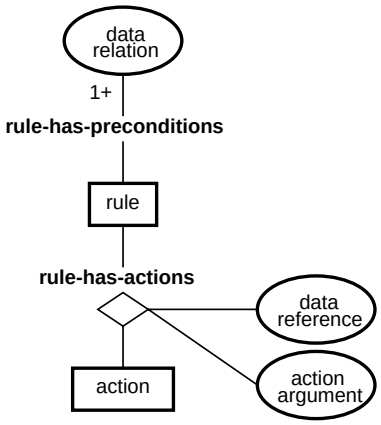


Figure 5.11: *Rule preconditions and actions*

Criteria The different criteria that have been mentioned so far can be represented by either a rule base or a function. In case that criteria have a rule-based representation, they can be seen as homogeneous sets of rules with a precise function with respect to the element which they are attached to, e.g. initialisation rule base of an operator.

Rule A rule consists of (see figure 5.11):

- preconditions or conjunction of logical formulas stating the conditions under which the rule fires (relation rule-has-preconditions). The formulas are similar to those in operator preconditions.
- actions which are performed when the rule is applied, which are modeled by the rule-has-actions relation connecting action, action argument and data reference. Although not all actions need an argument and/or a data reference, the former usually appears making reference to either an operator, or an operator argument or parameter.

There exist as many types of rules as different roles they play, i.e. initialisation, evaluation, repair, adjustment, and choice. The main differences between rule types reside in the allowed actions, which may vary from one engine to another, especially according to their repair capabilities:

- initialisation action, by simple assignment (`:=`) or interactively by the user (`initialise-by-user`).

- evaluation action, for which the possibilities are `assess-data`, `assess-parameter`, and `assess-operator`, and the same performed by the user (`assess-data-by-user`, `assess-parameter-by-user`, and `assess-operator-by-user`).
- repair action, for which the possibilities are re-executing the operator (`re-execute`), or transmitting the problem to a previous operator (with `send-up`, `send-down` or `send-op`) or choice point (`back-choice`).
- adjustment action, by an increase or decrease in the previous value (+= or -=), or directly by the user (`adjust-by-user`).
- choice action, with the options of using `use-operator`, `refuse-operator`, `use-operator-of-characteristic`, or `refuse-operator-of-characteristic`.

5.3.1.2 Additional concepts

The concepts necessary to define the PS task are presented next.

Operator knowledge base The operator knowledge base groups a set of operators, either compound or primitive ones, and possibly a set of goals. The operators describe the programs to be supervised, as well as the typical combinations of programs that serve to solve complex problems. The goals define the high-level processing functions that the operators can perform.

Problem specification The problem specification consists of the intended processing goal (modeled through a relation with functionality), the input data and the type required for the output data (relations ps-has-in-data and ps-has-out-data), the initial state or domain objects constituting the problem context (relation ps-has-initial-state), and the constraints which must hold in the solution state (relation ps-has-constraints). Figure 5.12 shows all these elements.

It is to note that PEGASE works on problems specified by means of the intended processing goal, whereas PULSAR is driven by the constraints describing the solution state to achieve.

Problem solution A problem solution consists of a plan, which is a set of operators solving the problem specification, plus the final state with the results of the execution of this plan. Different alternatives for plan representation are ordered, unordered or partially ordered sets of operators.

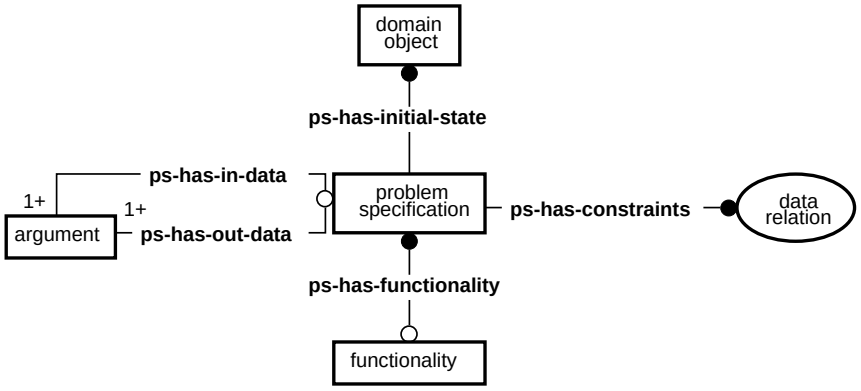


Figure 5.12: Problem specification

State and state history The state concept describes the execution context at a certain time. In PS the execution must be monitored to make sure that the plan has the expected results. If this is not the case, the cause must be determined and repaired either locally to the operator or by propagation of the problem to a previous point. This propagation implies backtracking to a previous state, thus giving rise to the need of keeping track of all of them in a state history.

5.3.2 Program supervision task and methods

Here we present the PS task and the methods that realise it in PEGASE, PULSAR and MedIA. These methods describe how the PS task is decomposed into subtasks, and so on. For the PS task, we show a top-level task decomposition which provides a global view of the PS method. For PS subtasks, we explain the methods that realise them and elaborate on the assumptions that they make on domain knowledge, if there are any. Complex methods are clarified by providing the inference structure and an algorithmic description of the control within it. The assumptions (labelled from *(a)* to *(u)*) are presented together with the method from which they have been identified.

In the graphical notation used in Kadstool to describe inference structures, ovals represent inference steps and boxes represent knowledge roles. Thick boxes represent static roles, which are knowledge roles that, without being modified by inferences, are necessary to perform them. Inference structures are networks showing how inference steps can be connected through knowledge roles, without defining control. When an inference step is further decomposed into

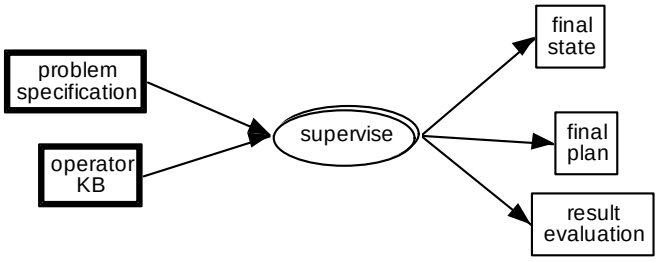


Figure 5.13: *Program supervision task*

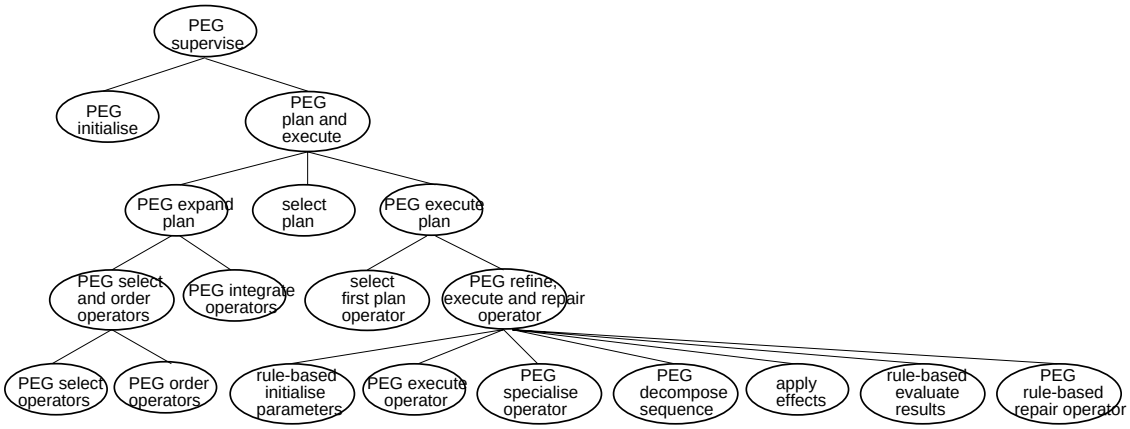


Figure 5.14: *Top-level task decomposition in PEGASE*

another inference structure, it is displayed as a double oval and its roles are marked in the decomposition.

The **PS task** receives a problem specification and the operator knowledge base as input, and produces a plan, a final state, and a result description indicating if the plan was successfully executed or not as output. Figure 5.13 depicts this.

The analysis of PEGASE and PULSAR methods, which has been carried out first, is presented next. The latter modeling of MedIA, limited to the top-level decomposition of the task, comes afterwards.

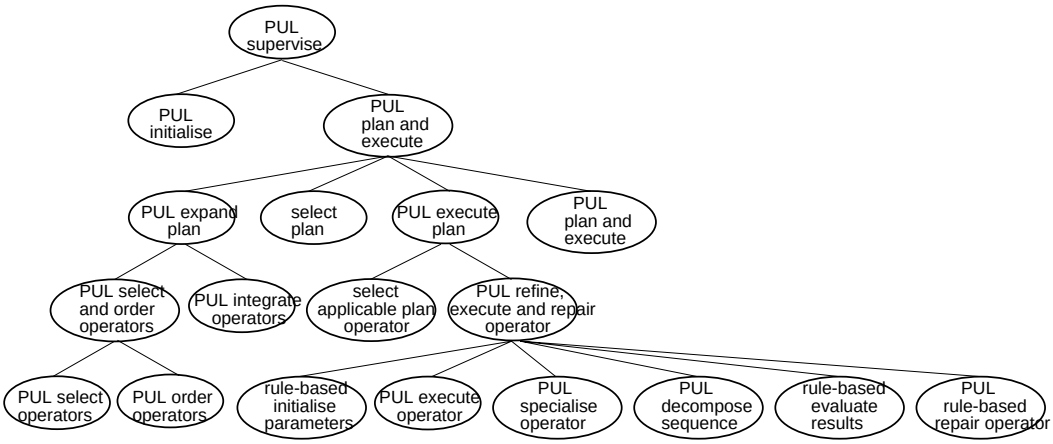


Figure 5.15: *Top-level task decomposition in PULSAR*

5.3.2.1 PEGASE and PULSAR methods

Figures 5.14 and 5.15 respectively show the top-level task decompositions of PEGASE and PULSAR PS methods. PS task is carried out by the sequential application of the steps initialise and plan and execute. The initialise step simply extracts the necessary information from the problem specification and creates an empty plan to start with. The plan and execute step is in charge of the main PS subtasks.

Next we concentrate on the methods for the plan and execute subtask and for the subtasks within it. We focus on those methods which are relevant to the definition of assumptions. The complete description of PEGASE and PULSAR subtasks can be found in appendix B.

Plan and execute This subtask constitutes the core of the operator-based planning. It consists of the steps expand plan, select plan, and execute plan. PEG plan and execute (in figure 5.16) first performs a plan expansion guided by the intended processing fuctionality. Then it searches for a plan that solves the problem, i.e. a plan that yields a successful execution result (result evaluation is success):

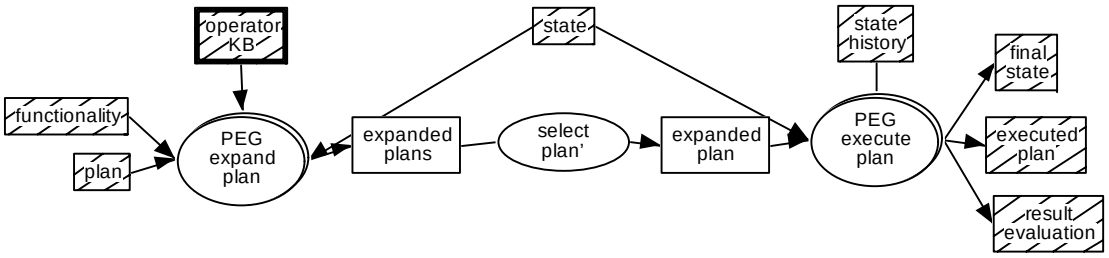


Figure 5.16: *Inference structure of* PEG plan and execute

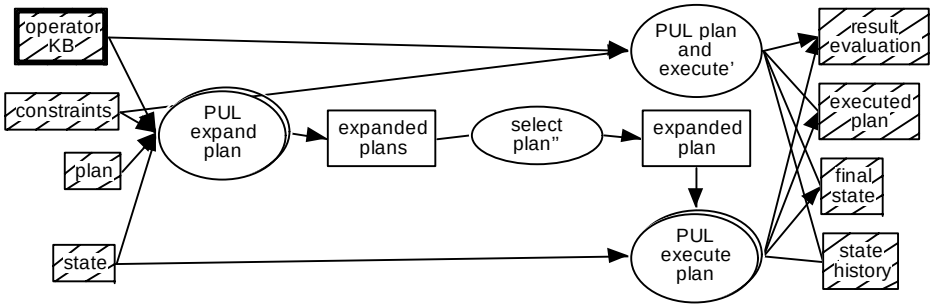


Figure 5.17: *Inference structure of* PUL plan and execute

PEG expand plan (plan+state+functionality+operator KB→ expanded plans)
while result evaluation <> success **and** expanded plans <> ∅ **do**
  select plan (expanded plans→ expanded plan+expanded plans)
  PEG execute plan (expanded plan+state+state history→ result evaluation+executed plan+final state+state history)
end while
if expanded plans = ∅ **then**
  result evaluation← all-plan-expansions-failed
end if

PUL plan and execute (in figure 5.17) expands plans guided by the constraints in the problem specification describing the solution state. Then it searches for a plan that is successfully executed but, unlike PEGASE, it tries every additional plan expansion (recursive call to PUL plan and execute) to make sure that no unsatisfied constraints remain:

```
if constraints ∈ state then
  executed plan← plan
  final state← state
  result evaluation← success
else
  PUL expand plan (plan+state+constraints+operator KB→ expanded plans)
  while result evaluation <> success and expanded plans <> ∅ do
    select plan (expanded plans→ expanded plan+expanded plans)
    PUL execute plan (expanded plan+state+state history→ result evaluation+executed
    plan+final state+state history)
    if result evaluation = success then
      PUL plan and execute (executed plan+final state+state history+constraints+operator
      KB→ result evaluation+executed plan+final state)
    end if
  end while
  if expanded plans = ∅ then
    result evaluation← all-plan-expansions-failed
  end if
end if
```

PEGASE only performs one plan expansion, under the assumption that individual operators contain enough information to solve a problem, or what amounts to the same thing, that *the operator knowledge base describes the entire set of possible solutions (a)*. PEGASE is therefore more suitable for knowledge bases containing only hierarchical operators since such operators represent a set of predefined solutions. PULSAR makes no assumption on the contents of the operator knowledge base, as it tries every possible operator combination.

Expand plan This subtask consists of the steps select and order operators and integrate operators. According to some heuristics, it selects the candidate operators from the operator knowledge base, orders them, and then integrates them in the current plan. The result is a set of expanded plans. This task is similar in both engines. The differences reside in the heuristics used for the select and order operators subtask and in the plan representation assumed by the integrate operators subtask.

Select and order operators Different heuristics can be used to determine the operators to be incorporated in the current plan and to order them. These differences are made explicit in the alternative methods that perform select operators and order operators. The used heuristics impose assumptions.

In PEG select operators an operator is selected when it fulfils the required processing function and when the types of all its input and output arguments match the input and output data types in the problem specification. PEG order operators gives precedence to

operators that use the maximum of inputs and outputs in the problem specification, and which are related to the goal solving the required function via the relation goal-achieved-by. PEGASE needs that *operators specify their functionality or else a goal defines it (b).*

In PUL select operators the selection is based on a matching between operator effects and problem specification constraints which neither the current state nor the expected effects of operators already in the plan have yet satisfied. PUL order operators uses a complex heuristic: maximum of outputs for which all effects can be achieved, maximum of inputs for which all preconditions hold, and operators that are higher in the abstraction hierarchy. PULSAR requires that *operators contain knowledge about their side-effects (c).*

Integrate operators The method to perform integrate operators depends on plan representation and hence makes assumptions on this aspect. In PEGASE *a plan is a single (hierarchical) operator (d)*, whereas in PULSAR *a plan is an unordered set of operators (e).* Although for different reasons, PEG integrate operators and PUL integrate operators simply incorporate the operator without any commitment on the order, so no more assumptions are made.

Select plan Once all possible plans have been expanded, one of them must be selected to be executed. This subtask can be performed in a more or less intelligent way. However, the method used in both engines simply consists in selecting the first plan. No assumptions are made.

Execute plan This subtask corresponds to the selection of an operator from the plan and its refinement and execution, that is, select plan operator plus refine, execute and repair operator. Since a plan is a single (hierarchical) operator, this process is repeated in PEGASE only once:

select first plan operator (plan+state→ unified operator)
PEG refine, execute and repair operator (unified operator+plan+state+state history→ result evaluation+executed plan+final state+state history)

In PULSAR all the operators in the plan must be refined and executed. Therefore, PUL execute plan proceeds along the operators in the plan, while the evaluation of the execution results is positive (result evaluation is success):

result evaluation← success
select applicable plan operator (plan+state→ unified operator)
while unified operator <> ∅ **and** result evaluation = success **do**
  PUL refine, execute and repair operator (unified operator+plan+state+state history→ result evaluation+executed plan+final state+state history)
  plan← executed plan
  state← final state
  select applicable plan operator (plan+state→ unified operator)
end while

Select plan operator The method in PEGASE for this subtask is trivial, again due to the kind of plans it deals with (select first plan operator), and thus makes no assumption. PULSAR performs this selection based on a matching between operator preconditions and the current state (select applicable plan operator). PULSAR assumes that *the operator knowledge base contains knowledge about preconditions (f).*

Refine, execute and repair operator This subtask constitutes the core of the hierarchical planning. It performs either the hierarchical refinement of compound operators or the execution of primitive ones, with the subsequent evaluation and repair steps. In PEG refine, execute and repair operator (see figure 5.18) first the operator parameters are initialised and then the operator is executed if it is a primitive one, specialised or decomposed otherwise. Afterwards the operator effects are used to update the state. After this, the results are evaluated and, if necessary, the operator is repaired. This evaluation-repair loop is repeated while the evaluation step yields an indication to repair (result evaluation is repair). The plan must be updated after every operator execution to indicate that an actual program has been applied:

rule-based initialise parameters (operator+state→ initialised operator)
if primitive operator **then**
  PEG execute operator (initialised operator+state+state history→ final state+state history)
  update plan (plan+initialised operator→ final plan)
else if specialisation **then**
  PEG specialise operator (initialised operator+plan+state+state history→ final plan+final state+state history)
else if sequence **then**
  PEG decompose sequence (initialised operator+plan+state+state history→ final plan+final state+state history)
end if
apply effects (initialised operator+final state→ final state)
rule-based evaluate results (operator+final state→ evaluated operator+result evaluation)
while result evaluation = repair **do**
  PEG rule-based repair operator (evaluated operator+final state+state history→ final state+state history)
  rule-based evaluate results (operator+final state→ evaluated operator+result evaluation)
end while

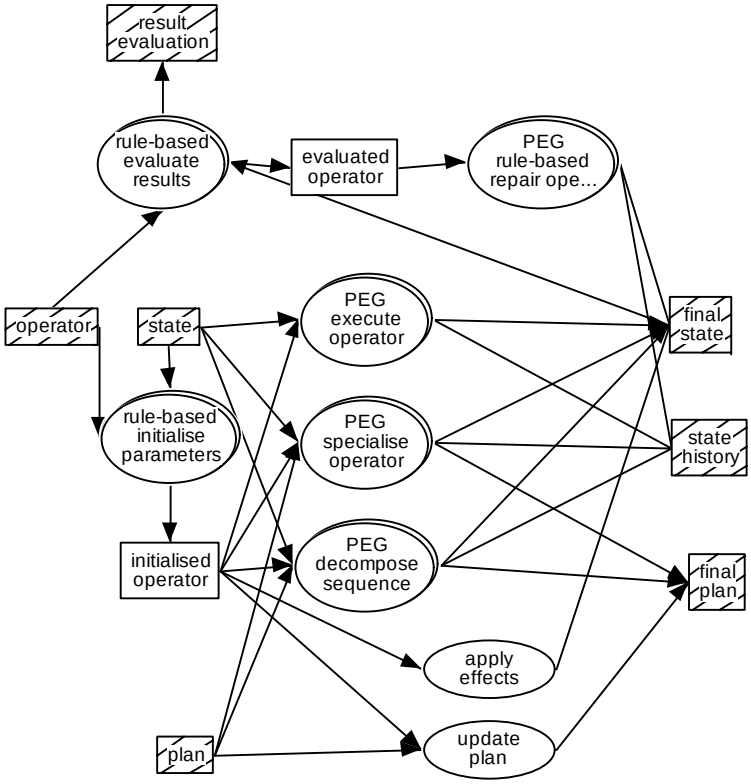


Figure 5.18: *Inference structure for* PEG refine, execute and repair operator

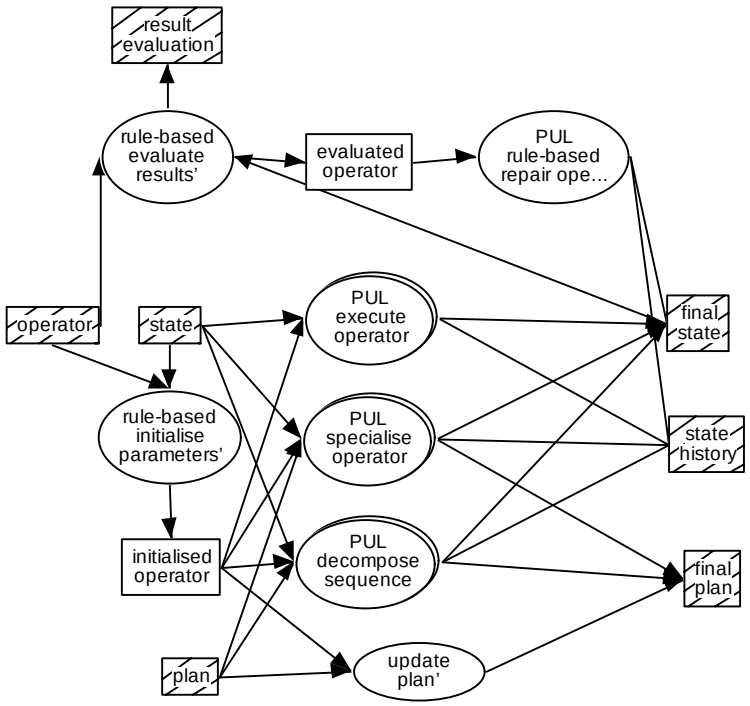


Figure 5.19: *Inference structure for* PUL refine, execute and repair operator

PUL refine, execute and repair operator (in figure 5.19) is similar, except for the different realisations of the main steps and for the absence of the apply effects step.

Initialise parameters The parameters of an operator must be given a value whenever it is going to be refined or executed. This is done by initialise parameters. For this step to be successful, the knowledge used for initialisation must be adequate to the operator characteristics. This means that *the initialisation criteria of the operator must give means to initialise all parameters without a pre-established value* (by default value, data distribution, or data flow) *(g)*.

The method used in both engines is rule-based initialise parameters, which consists in the application of a simple forward chaining[1] to the initialisation criteria. The first assumption it makes is that *the initialisation criteria follow a rule-based representation and that rules are*

[1] In this forward chaining usually no chaining among the rules takes place. The working cycle proceeds as follows: the current state is observed, then all rules matching the state are selected and their actions are executed.

single-layered, which means that their consequents correspond directly to initialisation actions *(h)*. It makes other assumptions to ensure that the contents of this rule base are suitable for the initialisation role that it plays: that *there is at least one rule for each parameter without value (i)*, and that *the rules initialising a parameter cover all possible situations (j)*.

Execute operator This subtask actually calls the program in the library and collects the execution results. PEG execute operator and PUL execute operator present differences but these are not relevant to the identification of assumptions.

Specialise operator This subtask corresponds to the refinement of specialisation decompositions. PEG specialise operator (in figure 5.20) obtains the candidates for the specialisation in the current state, and updates the state history indicating that a specialisation is going on. Then it sorts the candidates based on the number of times a suboperator has been chosen, and it treats the first one. To do this, the operator inputs corresponding to suboperator inputs are distributed, i.e. the connections specified in the data distribution are set (by distribute in arguments), and the suboperator is refined and executed. Afterwards the outputs are distributed to the corresponding operator outputs and the process ends. Finally the state history is updated:

rule-based choose suboperators (operator+state→ suboperators)
update hist with spec (operator+state history→ state history)
sort suboperators (suboperators→ suboperators)
next (suboperators→ suboperator)
distribute in arguments (operator+suboperator→ suboperator)
PEG refine, execute and repair operator (suboperator+plan+state+state history→ result evaluation+final plan+intermediate state+state history)
distribute out arguments (operator+intermediate state+final state→ final state)
update hist with exec (final state+state history→ state history)

PUL specialise operator (in figure 5.21) differs. It obtains the candidates for the specialisation given the current state, and updates the state history accordingly. Then it searches for a suboperator that can be successfully refined and executed. Before this, the operator inputs are distributed, and the suboperator preconditions are tested in order to detect runtime incompatibilities between the current state and suboperator requirements. In such cases the current suboperator is discarded and the next candidate is tried. In case of successful execution of any of the suboperators, the outputs are distributed and the process ends. The other end condition is that all suboperators have been tried without result. The state history is updated at the end:

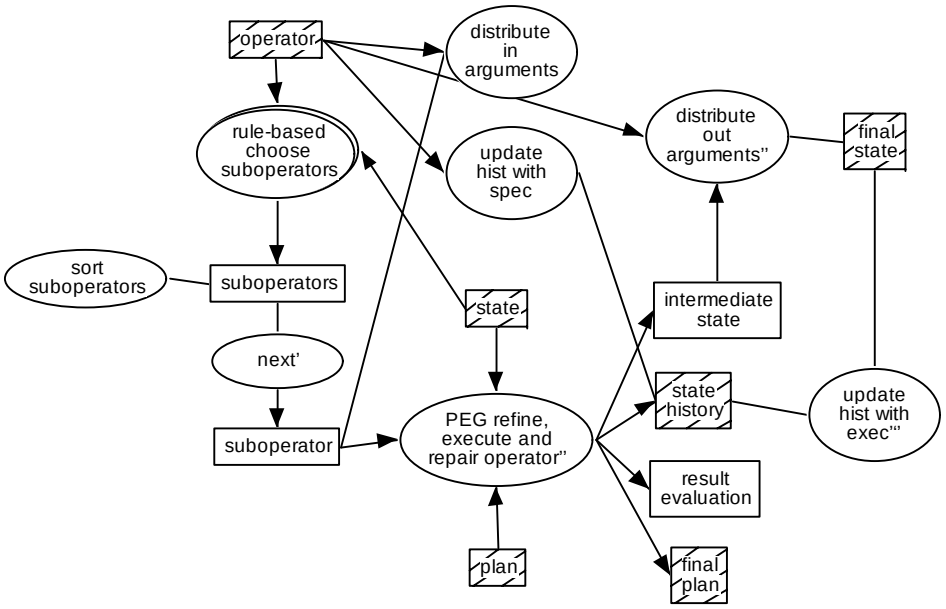


Figure 5.20: *Inference structure for* PEG specialise operator

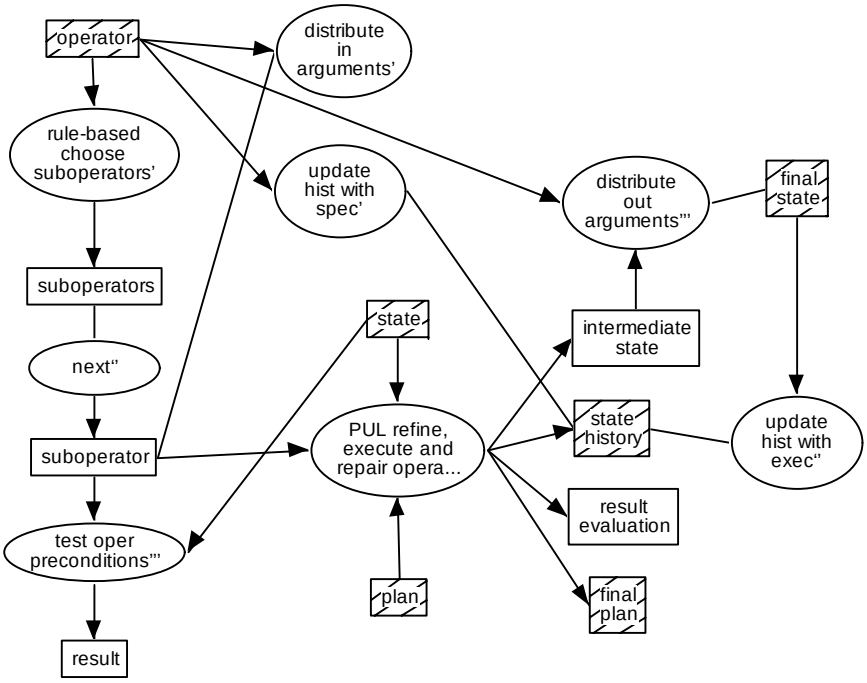


Figure 5.21: *Inference structure for* PUL specialise operator

```
rule-based choose suboperators (operator+state→ suboperators)
update hist with spec (operator+state history→ state history)
intermediate state← ∅
repeat
  next (suboperators→ suboperator)
  distribute in arguments (operator+suboperator→ suboperator)
  test oper preconditions (suboperator+state→ result)
  if result = success then
    PUL refine, execute and repair operator (suboperator+plan+state+state history→ result
    evaluation+final plan+intermediate state+state history)
    if result evaluation = continue then
      distribute out arguments (operator+intermediate state+final state→ final state)
    end if
  end if
until result evaluation = continue or suboperators = ∅
update hist with exec (final state+state history→ state history)
```

PEGASE refines and executes the best candidate in terms of the number of times it has been chosen. PULSAR, nevertheless, exhaustively searches for a suboperator that can be successfully refined and executed in the set of candidates. PULSAR assumes that *the choice criteria may select several candidates (k)*, and thus it is more suitable for such kind of choice criteria.

Choose suboperators The step choose suboperators selects the candidates for a specialisation in the current situation. The knowledge used for this purpose should be adequate to the specialisation. This means that *the choice criteria associated to the specialisation should permit the choice of every suboperator in the specialisation (l).* The method used in both engines is rule-based choose suboperators. As in the case of rule-based initialise parameters, this method consists in the application of a simple forward chaining to the choice criteria. Therefore it assumes that *the choice criteria follow a single-layered rule-based representation (m).* Like rule-based initialise parameters, it makes other assumptions to ensure that the rule base contents are adequate to its choice role. As they are studied in chapter 6 as properties that PS knowledge bases should verify, they have been omitted here.

Decompose sequence This subtask corresponds to the refinement of sequential decompositions. PEG decompose sequence is shown in figure 5.22. First the suboperators in the sequence are obtained and the state history is updated indicating that a sequential decomposition is going on. Then every suboperator is sequentially treated. For this purpose, the inputs of the operator or the outputs of previous suboperators needed as inputs are distributed (by bind in arguments). If the suboperator is optional and its applicability criteria indicate that it

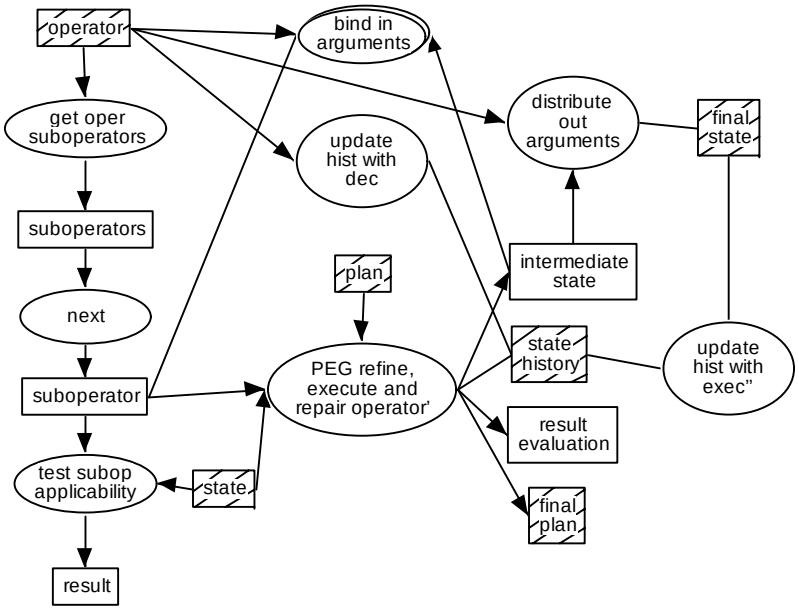


Figure 5.22: *Inference structure for* PEG decompose sequence

must not be used, it is skipped. Otherwise, the suboperator is refined and executed, like the operators that are not optional. Afterwards the outputs are distributed to the corresponding operator outputs and the next suboperator is treated. The process ends when one execution yields a negative result evaluation or when all suboperators have been treated. The state history is then updated:

get oper suboperators (operator→ suboperators)
update hist with dec (operator+state history→ state history)
intermediate state← ∅
repeat
  next (suboperators→ suboperator)
  bind in arguments (operator+suboperator+intermediate state→ suboperator)
  if optional **then**
    test subop applicability (suboperator+state→ result)
  end if
  if not optional **or** result = success **then**
    PEG refine, execute and repair operator (suboperator+plan+state+state history→ result evaluation+final plan+intermediate state+state history)
    distribute out arguments (operator+intermediate state+final state→ final state)
    state← final state
    plan← final plan
  end if
until result evaluation <> continue **or** suboperators = ∅
update hist with exec (final state+state history→ state history)

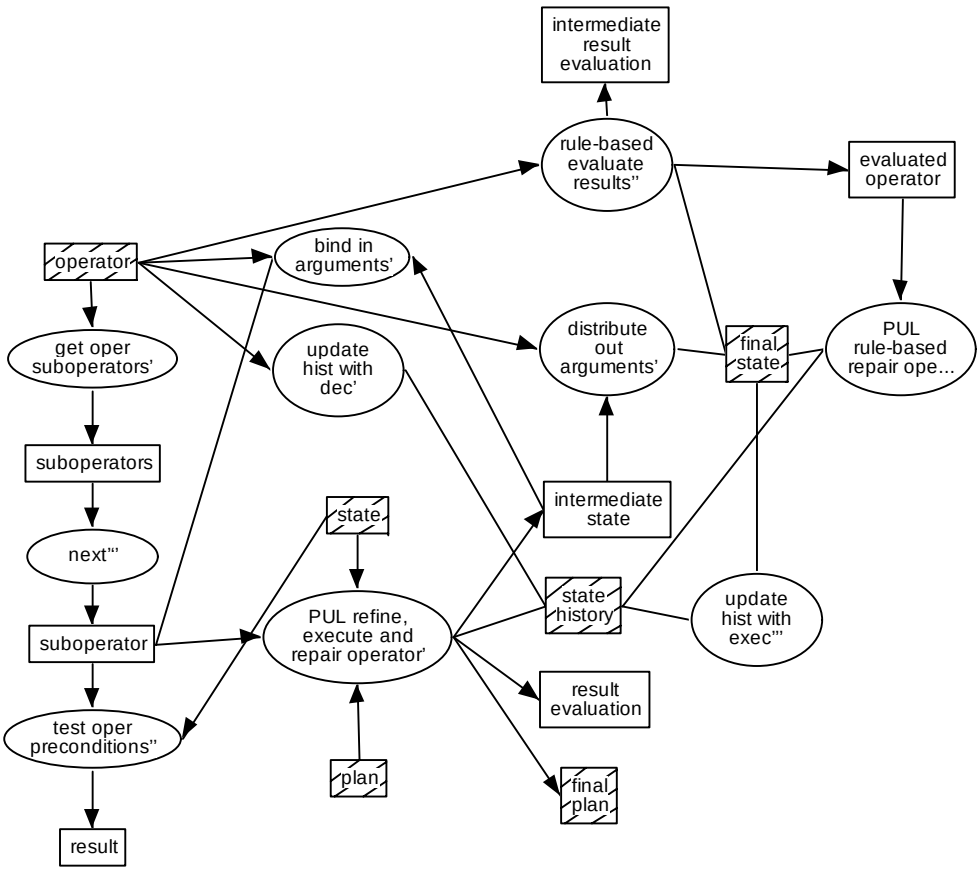


Figure 5.23: *Inference structure for* PUL decompose sequence

PUL decompose sequence (in figure 5.23) differs. First the suboperators are obtained and the state history is updated. Then every suboperator is sequentially treated. For this purpose the necessary inputs of the operator or outputs of previous suboperators are distributed. But in PULSAR suboperator treatment only takes place if its preconditions hold, to detect any run-time incompatibility. In that case the suboperator is refined and executed, and afterwards the outputs are distributed. Next the intermediate results are evaluated (by rule-based evaluate results), and if necessary the suboperator is repaired. Then the next suboperator is treated. The process ends when the preconditions of one of the suboperators do not hold, when one execution yields a negative intermediate result evaluation, or when all suboperators have been treated. The state history is then updated:

get oper suboperators (operator→ suboperators)
update hist with dec (operator+state history→ state history)
intermediate state← ∅
repeat
  next (suboperators→ suboperator)
  bind in arguments (operator+suboperator+intermediate state→ suboperator)
  test oper preconditions (suboperator+state→ result)
  if result = success **then**
    PUL refine, execute and repair operator (suboperator+plan+state+state history→ result evaluation+final plan+intermediate state+state history)
    distribute out arguments (operator+intermediate state+final state→ final state)
    rule-based evaluate results (operator+final state→ evaluated operator+intermediate result evaluation)
    while intermediate result evaluation = repair **do**
      PUL rule-based repair operator (operator+final state+state history→ final state+state history)
      rule-based evaluate results (operator+intermediate state→ evaluated operator+intermediate result evaluation)
    end while
    state← final state
    plan← final plan
  end if
until result <> success **or** intermediate result evaluation <> continue **or** suboperators = ∅
update hist with exec (final state+state history→ state history)

PULSAR performs an intermediate evaluation-repair loop in the course of sequential decompositions and thus it assumes that *the operator has knowledge about the required performance of the individual suboperators in the sequence and about how to repair potential problems in the sequence (n).*

Evaluate results The step evaluate results checks the results of operator refinement and execution in order to detect and diagnose potential problems. For this purpose *the evaluation criteria should give means of determining whether the results are acceptable or not (o).* Both engines use rule-based evaluate results, which implies the application of a simple forward chaining to the evaluation criteria of the operator. It assumes that *the evaluation criteria have a single-layered rule-based representation (p).* Other assumptions are made on the rule base contents related to its role are studied in chapter 6.

Repair operator The step repair operator is performed in case evaluate results detects a problem. It first obtains a corrective action to solve the diagnosed problem. Then either a re-execution (with the corresponding parameter adjustment) or the appropriate repair subtask is activated depending on the obtained action. This repair subtask in general propagates

the problem to another operator and forces its repair. Problem propagation implies that a chaining of repair steps may take place.

PEGASE and PULSAR differ in the repair subtasks that they incorporate. PEGASE considers the local repair of primitive operators, which implies the adjustment of operator parameters and a further execution, as well as the non-local repair, which consists in the propagation of the problem to another point (any operator or a previous choice point) with the activation of the repair of the corresponding operator. PULSAR considers the local repair of primitive operators and the non-local repair through the propagation of the problem to any of the suboperators.

Since repair operator treats the problems diagnosed by the step evaluate results, an important assumption is that *the repair criteria of an operator must give means to solve the problems diagnosed by the evaluation criteria of the operator (q)*. Moreover, *the repair knowledge of the operator must give means to solve any problem that it could receive from the repair knowledge of any other operator (r)*. A different assumption concerns *the correctness of the chaining of repair actions in case of problem propagation (s)*, e.g. it must end with an action to solve the problem.

The repair of the operator is carried out in both engines by a rule-based method: PEG rule-based repair operator and PUL rule-based repair operator. These methods perform an even more simple forward chaining[2] on the repair criteria of the operator. Both assume that *the repair criteria have a single-layered rule-based representation (t)*. They make other assumptions on the rule base contents related to its role which are analysed in chapter 6.

Adjust parameters The step adjust parameters is performed whenever the local repair of a primitive operator is required. Both engines use rule-based adjust parameters. As in rule-based initialise parameters, it consists in the application of a simple forward chaining to the adjustment criteria of the operator. It thus assumes that *the adjustment criteria follow a single-layered* rule-based representation *(u)*. It also makes other assumptions on the contents of this rule base that are studied in chapter 6.

[2]In this forward chaining no chaining among the rules takes place either. The working cycle differs from the previously explained: the current state is observed and then a single rule matching the state is selected and executed.

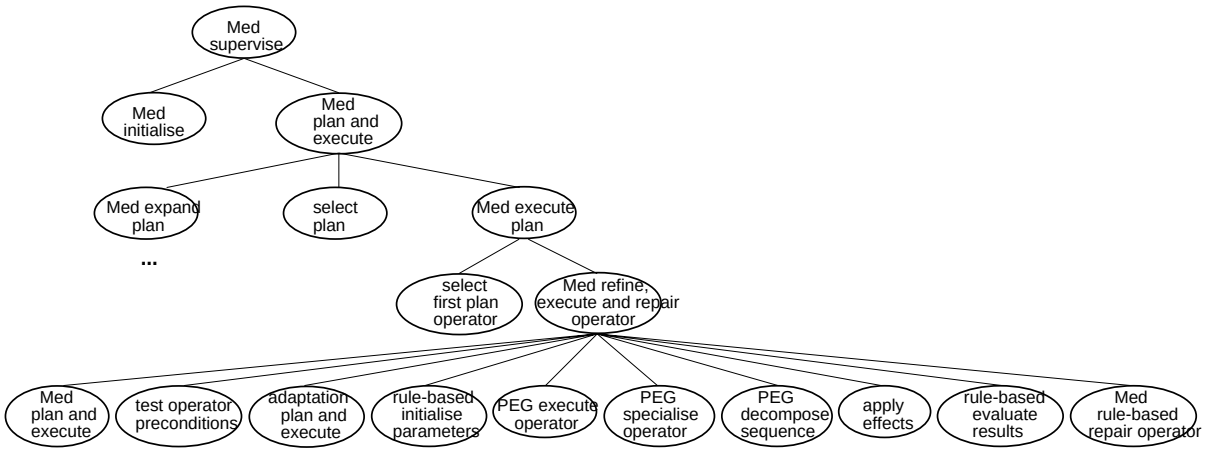


Figure 5.24: *Top-level task decomposition in MedIA*

5.3.2.2 MedIA methods

The top-level task decomposition of MedIA is presented in figure 5.24. In MedIA, PS task is also carried out by the steps initialise and plan and execute. Next we focus on the methods that perform the planning-related subtasks, i.e. plan and execute and refine, execute and repair operator subtasks.

Plan and execute The operator-based planning is similar to the one in PEGASE. Med plan and execute performs a plan expansion and then it searches for a plan (a hierarchical operator) that yields a successful execution result:

Med expand plan (plan+state+functionality+operator KB→ expanded plans)
while result evaluation <> success **and** expanded plans <> ∅ **do**
  select plan (expanded plans→ expanded plan+expanded plans)
  Med execute plan (expanded plan+state+state history→ result evaluation+executed plan+final state+state history)
end while
if expanded plans = ∅ **then**
  result evaluation← all-plan-expansions-failed
end if

Med expand plan is guided by specific attributes describing the data contents, in addition to the intended processing functionality and the input and output data types that PEGASE uses. Med select plan simply selects the first plan. Then, Med execute plan takes the only operator in the plan and refines it and executes it. The latter corresponds to the next subtask.

Refine, execute and repair operator MedIA integrates new capabilities concerning hierarchical planning. It permits abstract steps within sequential decompositions representing an abstract functionality to solve. Besides, it manages small run-time incompatibilities between the current state and (sub)operator preconditions.

Med refine, execute and repair operator, if the current step is abstract, performs a new operator-based planning step through a recursive call to Med plan and execute. Otherwise operator preconditions are tested in order to detect small run-time incompatibilities. If necessary, they are solved through a call to adaptation plan and execute, which performs an operator-based planning step driven by the unfulfilled preconditions. The rest of the steps are similar to those in PEGASE:

if abstract step **then**
  get oper functionality (operator→ functionality)
  Med plan and execute (plan+state+state history+functionality+operator KB→ result evaluation+final plan+final state)
else
  test oper preconditions (operator+state→ result)
  if result <> success **then**
    get oper preconditions (operator→ preconditions)
    adaptation plan and execute (plan+state+state history+preconditions+operator KB→ result evaluation+plan+state)
  end if
  rule-based initialise parameters (operator+state→ initialised operator)
  if primitive operator **then**
    PEG execute operator (initialised operator+state+state history→ final state+state history)
    update plan (plan+initialised operator→ final plan)
  else if specialisation **then**
    PEG specialise operator (initialised operator+plan+state+state history→ final plan+final state+state history)
  else if sequence **then**
    PEG decompose sequence (initialised operator+plan+state+state history→ final plan+final state+state history)
  end if
  apply effects (initialised operator+final state→ final state)
  rule-based evaluate results (operator+final state→ evaluated operator+result evaluation)
  while result evaluation = repair **do**
    Med rule-based repair operator (evaluated operator+final state+state history→ final state+state history)
    rule-based evaluate results (operator+final state→ evaluated operator+result evaluation)
  end while
end if

Contrarily to PEGASE, MedIA does not assume that the entire set of possible solutions has been described in the operator knowledge base, but rather that it may include solutions which have not been completely described or which need to be adapted at run time.

5.3.3 Assumptions of program supervision methods

The assumptions presented before, which are summarised in figure 5.25, are typical knowledge requirements of PS methods. Without the aim of being exhaustive, we have identified a collection of assumptions describing the knowledge that PS methods need, useful for V&V purposes. Nevertheless, we are aware that the set of assumptions that a PS method makes is potentially infinite, and that they cannot completely be isolated.

The assumptions have been informally identified from the information on the knowledge utilisation provided by the different (sub)task decomposition descriptions. Particularly:

- each (sub)task determines the precise role of the knowledge it uses and enforces general assumptions.
- the detailed description of the method that performs the (sub)task often restates the general assumptions of the (sub)task. It usually imposes additional assumptions. They can make reference to any requirement so that the method works properly, e.g. to the required knowledge or to the representation required for this knowledge. They can also identify which characteristics are more suitable according to method functioning.

For example, initialise parameters assumes that the initialisation criteria give an initial value to all operator parameters without a pre-established value *(g)*. The method rule-based initialise parameters requires that the initialisation criteria have a single-layered rule-based representation *(h)*, at the same time as it restates the general assumption taking into account this representation, that is, *(i)* and *(j)*.

The conceptual organisation for assumptions in [Benjamins and Pierret-Golbreich, 1996] distinguishes epistemological, pragmatic and teleological assumptions. The assumptions that we have identified correspond to epistemological ones. These are further divided into *availability* and *property* assumptions. In addition, in [Marcos et al., 1997] we proposed to distinguish the assumptions that are critical for the proper functioning of the method (*compulsory*) from those that are only advisable (*desirable*). Besides, the property assumptions can make reference to the knowledge *structure* or to the knowledge *role* imposed by the method.

In PS methods we find mainly:

- assumptions on availability of knowledge to perform certain subtasks (availability assumptions). Examples related to the use of selection/ordering heuristics are *(b)*, *(c)* and

(a)	the operator knowledge base describes the set of possible solutions
(b)	operators specify their functionality or a goal defines it
(c)	operators specify their side-effects
(d)	plans are single (hierarchical) operators
(e)	plans are unordered sets of operators
(f)	operators specify their preconditions
(g)	initialisation criteria give means to initialise all parameters without value
(h)	initialisation criteria are single-layered rule bases
(i)	initialisation rule bases contain at least one rule for each parameter without value
(j)	rules initialising a parameter cover all possible situations
(k)	choice criteria may select several candidates
(l)	choice criteria permit the choice of every suboperator in the specialisation
(m)	choice criteria are single-layered rule bases
(n)	sequential compound operators have evaluation and repair criteria concerning the suboperators in the sequence
(o)	evaluation criteria give means of determining whether the results are acceptable or not
(p)	evaluation criteria are single-layered rule bases
(q)	repair criteria give means to solve the problems diagnosed by the evaluation criteria of the same operator
(r)	repair criteria give means to solve any problem transmission that could be received from the repair criteria of any other operator
(s)	chains of repair actions are continuous and end with an action that can solve the problem
(t)	repair criteria are single-layered rule bases
(u)	adjustment criteria are single-layered rule bases

Figure 5.25: *Assumptions of program supervision methods*

(f). A different example is *(n)*, which concerns the availability of knowledge to perform an intermediate evaluation-repair loop within sequential decompositions.

- assumptions about the required knowledge representation (property assumptions). See *(d)* and *(e)* as examples related to plan representation and *(h)* as example concerning criteria representation.
- assumptions on other knowledge characteristics, critical or not (compulsory or desirable property assumptions). An example of compulsory property is *(g)*, which describes the required characteristics of the initialisation criteria according to their role. An example of desirable property is *(a)*, which states that the method is suitable for hierarchical operators.

5.4 Additional benefits of knowledge modeling

The initial motivation of our knowledge modeling was studying the organisation of the knowledge that the PS methods require, and how they use this knowledge to solve the PS task. This analysis is the key to determining the assumptions that PS methods make on domain knowledge. Indeed, the assumptions of a PS method specify what it needs to operate and thus constitute important properties that the knowledge base should verify [Marcos et al., 1997]. Our knowledge modeling has other interesting outcomes [Marcos et al., 1998a], particularly derived from different utilisations of the descriptions of PS methods and their assumptions.

5.4.1 Benefits from the description of program supervision methods

The task decompositions corresponding to PEGASE, PULSAR and MedIA methods constitute high-level functional specifications and thus can be used to support the manual V&V of PS engines by inspection.

In addition to this, PEGASE, PULSAR and MedIA task decompositions can be of help for the design of new PS engines. Indeed the design of MedIA has been carried out by comparison with PEGASE and PULSAR task decompositions, reusing part of their methods [Crubézy et al., 1998].

Figure 5.26 synthesizes the top-level PS methods using a task-method decomposition structure similar to the one used in [Orsvärn, 1996], where a method consists of subtasks, and a

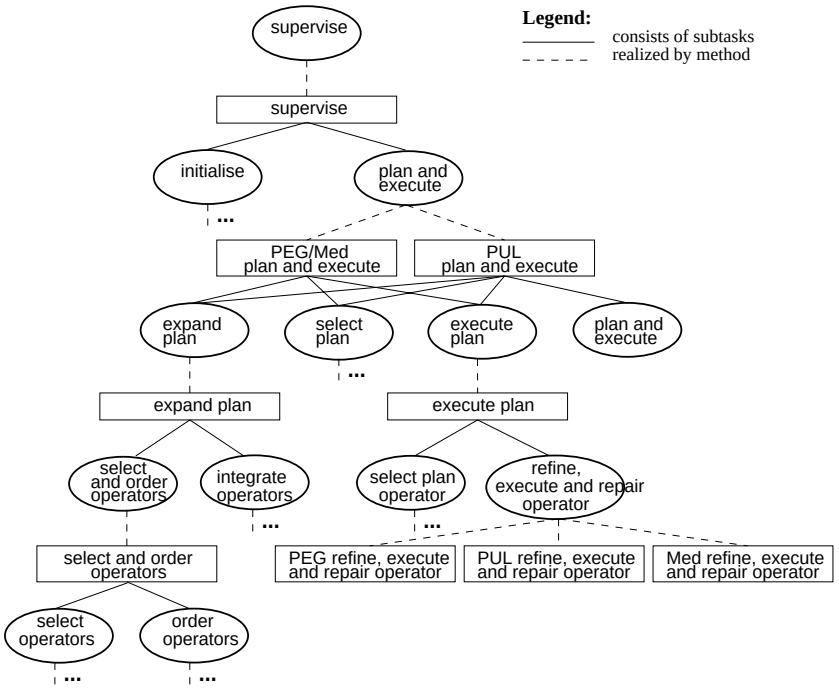


Figure 5.26: *A task-method decomposition for PS*

(sub)task can be realised by alternative methods. The PEGASE (and MedIA) and PULSAR methods for plan and execute are different: PUL plan and execute is recursive whilst PEG/Med plan and execute is not. Differences can be found in methods for other subtasks at a lower level. The PS task-method decomposition can be seen as a high-level specification of different PS methods that could be used to configurate new PS engines, e.g. by changing the control of a given method or by incorporating new steps.

Finally, PEGASE, PULSAR and MedIA task decompositions have served to identify typical methods that perform certain subtasks. For instance, the methods for specialise operator and decompose sequence in MedIA are similar to the equivalent ones in PEGASE. This has put forward the interest of including in the BLOCKS library other usable components corresponding to such typical methods. These new BLOCKS components, different from the current small-grain sized ones, would allow a more efficient reuse for less-skilled engine designers.

5.4.2 Benefits from the description of assumptions

The obtained assumptions provide a characterisation of the studied PS engines in terms of their knowledge requirements. They can be used to determine the adequacy of a PS engine to a target domain, as in [Benjamins et al., 1996b]. For instance, the assumption *(a)* of PEG plan and execute implicitly states that PEGASE handles hierarchical operators so it is suitable for domains in which operator functions are naturally seen as combinations of actions. This is an important issue for engineering new applications, because the success of the final application will often depend more on the adequacy of the selected engine to the features of the domain rather than on the efficiency of the algorithms that implement it [Nunes de Barros et al., 1996].

In addition, we have obtained not only a description of the assumptions of the methods that PEGASE, PULSAR and MedIA implement, but also of the different methods that they use to perform the PS subtasks at different levels. Jointly with the previous task-method decomposition for PS, they could be used to define the knowledge base verification issues for new PS engines.

5.5 Related work

For the description of the PS methods in this chapter, we have used the perspective of the knowledge modeling of PULSAR in [van den Elst, 1996] as starting point. Here the objective is obtaining a functional specification to be used as a basis for the implementation of the PULSAR engine. On the other hand, our analysis goes further into the details about knowledge utilisation by the engine. We have also made an effort to integrate the descriptions of PULSAR, PEGASE and MedIA engines under the same modeling perspective. This integrated view has several benefits, not only for V&V purposes but also for knowledge acquisition ones, as we have shown in the previous section. Next we cite some work which is closer to our purposes.

In [Valente, 1994] we find a knowledge-level analysis of classical planning systems in terms of the types of knowledge they use and how they are structured. The aim is to provide a framework for classifying methods in order to facilitate their selection for a particular application. This work is complemented in [Nunes de Barros et al., 1996] with an analysis of the planning subtasks and the use that they make of the different types of knowledge. Although with a different initial motivation, our knowledge analysis comprises most of these aspects.

A study of the assumptions of a group of problem-solving methods for model-based diagnosis is presented in [Fensel and Benjamins, 1996]. Assumptions are viewed here as a characterisation of problem-solving methods and as guidelines for the acquisition process of domain knowledge. Our study of assumptions is along this line but with V&V purposes.

Chapter 6

Verification of Program Supervision Knowledge Bases

The knowledge modeling of PEGASE, PULSAR, and MedIA engines provides us with a description of the PS domain model and methods (including the methods for PS subtasks), and of some assumptions that methods make. We have used these elements to define the properties that a PS knowledge base should verify to adequately serve the reasoning of the intended engine. Taking into account these properties and the target knowledge representation, we have implemented a new Lama module for the model-based verification PS knowledge bases.

In this chapter we first present the model-based verification properties that we have defined from the PS model and methods. Then we describe the knowledge representation in Lama, and the implementation-dependent techniques that have been chosen accordingly. We next describe the issues for the integration of knowledge base verification in Lama and present the verification modules that have been implemented in this light. Finally we report on some experiences in the application of the verification modules to two different real-world knowledge bases and end with some conclusions.

6.1 Knowledge base verification properties

We have used the descriptions in chapter 5 to define typical properties that a PS knowledge base should verify according to the knowledge requirements of PEGASE, PULSAR and Me-

dIA. These properties correspond to the knowledge characterisation given by the PS domain model and to the information on the knowledge utilisation provided by the description of the PS methods of these engines. First, the PS domain model defines the necessary concepts in the knowledge base, and their structure, and establishes the necessary interrelations. Second, PS subtasks, especially the most knowledge demanding ones, precisely determine the role of the knowledge they use and consequently impose general assumptions. Third, the description of the method that performs the subtasks (inference structure plus control over the steps) makes clear the knowledge requirements so that the method works properly, or which knowledge characteristics, though not critical, are more appropriate. It often refines the role assumptions. In this section we describe the former properties and revise the latter ones, which have been addressed in chapter 5 as the assumptions of PS methods.

The verification properties can be classified according to different points of view. Many of them are characteristics common to all PS methods or *general* characteristics, but there also exist *method-specific* ones which arise when we consider the behaviour of a particular PS method. As assumptions, they can make reference to the knowledge *structure* or to the knowledge *role* imposed by the PS method. The latter are derived from the role that each piece of knowledge plays in the reasoning of PS. Finally, the properties can be critical (*compulsory*) or not (*desirable*).

Roughly speaking, we can say that the properties concerning the knowledge structure correspond to general characteristics, whereas the properties concerning the knowledge role are method-specific ones. However, there are properties related to the knowledge structure which are imposed by the functioning of a particular method.

In the following we present the properties a knowledge base for PS must comply with. We have grouped them in accordance with their degree of generality with respect to PS methods (general characteristics versus method-specific ones).

6.1.1 General properties of program supervision methods

General properties comprise structural properties derived from the PS domain model as well as role properties enforced by the main PS subtasks. Since role properties are refined by the description of the particular method that performs the subtask, they are treated in subsection 6.1.2.

Regarding structural properties, their definition needs a detailed study of the PS domain model in order to describe the syntactic properties of basic concepts in the knowledge base

like operators and goals, and also of more complex elements such as the operator knowledge base. Other properties are derived from the semantics of certain relationships.

6.1.1.1 Structural properties

Goal Goals must have a functionality, and at least one input argument and one output argument. They may have a set of constraints describing the expected final state.

A goal is related to the operators that achieve its functionality. To be adequately associated, these operators must use all the input arguments of the goal to produce (at least) the required output arguments. The compatibility in the number and data type of arguments between the goal and the related operators is necessary:

- the number of input arguments should be equal to the number of input arguments in the goal.
- the number of output arguments should be equal or greater than the number of output arguments in the goal.
- the data type of input (output) arguments should be the same as the corresponding goal argument, or a subtype.

Operator Operators must have at least one input argument and one output argument, but they may have or may not have any parameter. The functionality and characteristics of operators are not compulsory in principle. Operators may have one or more preconditions and one or more effects, and possibly one or more postconditions.

Operators with parameters need knowledge to initialise them, either through default values, data distribution, data flow, or initialisation criteria. Knowledge about how to evaluate their execution results and detect and diagnose problems, and about how to repair the possible problems is optional in principle.

Primitive operator Primitive operators should contain information about the calling syntax of the library program they represent.

Compound operator Compound operators have to be associated to a decomposition consisting of two or more suboperators. Choice decompositions need some choice criteria to select the candidate(s) for the specialisation at execution time.

Compound operators must also specify data distribution. Data flow specification is mandatory in sequential decompositions. The specification of data distribution describes how operator input arguments and parameters are connected to the input of suboperators, and how suboperator output arguments are connected to the output of the operator. The specification of data flow describes how output arguments of suboperators are connected to the input of other suboperators in the sequence.

In expressions of the form $\langle operator_i \rangle.\langle argument_j \rangle \rightarrow \langle operator_k \rangle.\langle argument_l \rangle$, which are used for the specification of data distribution and data flow, the permitted references vary with the intended use. For instance, in the specification of input distribution, the term $\langle operator_i \rangle.\langle argument_j \rangle$ can be either an input argument or a parameter of the operator, and $\langle operator_k \rangle.\langle argument_l \rangle$ must be a compatible input of any of the suboperators. This compatibility is established in terms of the kind and data type of inputs:

- the kind of inputs (argument or parameter) must be the same.
- the data type of inputs must be the same, or a subtype.

In the specification of data flow, both $\langle operator_i \rangle$ and $\langle operator_k \rangle$ must be suboperators belonging to the decomposition, and $\langle argument_j \rangle$ and $\langle argument_l \rangle$ must be compatible arguments, output and input, respectively, of $\langle operator_i \rangle$ and $\langle operator_k \rangle$.

A compound operator is related to a combination of other operators that serves to perform its functionality, transforming its input arguments into the specified output arguments. In general, in the specification of data distribution, all operator inputs must be used and all operator outputs must receive a value. Otherwise either the operator specifies more inputs than necessary or the suboperators do not provide enough outputs for the operator. On the other hand, not necessarily all suboperator outputs must be used by the operator, since they may include some needed to perform other processing functions.

In specialisation decompositions (*data distribution correctness in specialisations*):

- all operator inputs must be used by at least one suboperator.
- all operator outputs must receive one value from each suboperator.

- all suboperator inputs must receive one value.

In sequential decompositions (*data distribution correctness in sequences*):

- all operator inputs must be used by at least one suboperator.
- all operator outputs must receive one value at most.

Notice that not all suboperator inputs must receive a value through data distribution, since it can be established by means of data flow as explained below.

In sequential decompositions, the specification of data flow must provide a value to all suboperator inputs that are not considered in data distribution (*data flow correctness*):

- all suboperator inputs which do not correspond to any operator input (i.e. which do not appear in data distribution) must receive one value from the output of any other suboperator.

Hierarchical operator A compound operator is described at different levels of abstraction by means of decompositions, constituting a hierarchical operator. At some point the decompositions should end with primitive operators, otherwise the compound operator would not describe a combination of library programs. Therefore hierarchical operators must form a cycle-free tree structure where the leaves correspond to primitive operators (*hierarchical operator correctness*).

Operator knowledge base Operators are grouped into an operator knowledge base in which there may be included a set of goals describing the operators at the most abstract level. If such is the case, the goal set should cover all the functionalities of the operators in the knowledge base at the highest level of abstraction (*completeness of the goal set with respect to the abstract functionalities in the operator knowledge base*). This may vary from the functionality of hierarchical operators to the functionality of primitive operators when they do not appear in any hierarchy.

Criteria Most of the times criteria are fulfilled by rule bases with a particular role with respect to the element which they are attached to (operator, decomposition or operator within a decomposition).

Rule bases should be free of potential sources of errors such as redundancy. The analysis of critical errors that can prevent rule bases from adequately performing their role is more important. Examples are conflicting rules i.e. rules concluding conflicting actions in the same situation, or gaps i.e. situations that are not covered by any rule. However this analysis takes a better place in the context of the role played by the rule base because this precisely determines the necessary contents.

Rule Rules are typed according to their role (initialisation, evaluation, repair, adjustment, and choice) and this determines the action types allowed, and also the elements that can be referenced. In general, preconditions and actions cannot make reference to elements other than the associated operator, its arguments and parameters, and domain objects constituting the execution context. Regarding preconditions:

- repair preconditions are the only ones allowed to inquiry about the assessment of the operator.

Concerning actions, we may find assignments to arguments or domain objects in any type of actions. Other characteristics specific to the different action types follow:

- initialisation and adjustment actions are the only ones permitted to assign a value to operator parameters.
- choice actions should only make reference to suboperators in the decomposition.
- evaluation actions are allowed to assess either the operator itself or one of its parameters or output arguments.
- repair actions determine the operators that can be referenced: the superoperator (with `send-up`), one of the suboperators (with `send-down`), or any other operator (with `send-op`).

6.1.2 Specific properties of program supervision methods

In this part we present the properties specific to particular PS methods (including methods for PS subtasks). For this purpose we revise the methods that, according to our knowledge

model, most frequently realise the main PS subtasks, and their assumptions. In this regard we distinguish: the specific knowledge that methods require, the role properties that they clarify, and other non-critical properties, i.e. those more appropriate to their functioning.

6.1.2.1 Structural properties

Different method-specific structural properties have been identified. Some of them correspond to certain engine specifics, whilst others refer mainly to the use of complex heuristics in certain subtasks.

To illustrate the former we cite a characteristic of PEGASE and MedIA, which allow optional steps in sequential decompositions. In this case, the optional suboperator must be associated to some criteria to determine its applicability in the decomposition or applicability criteria. The possibility of optional suboperators also relaxes the requirement for data distribution correctness as it has been described before, since operator outputs may receive a value from more than one suboperator (*data distribution correctness in sequences (with optional operators)*).

As examples of the latter, smart selection or ordering mechanisms use heuristics that usually demand additional knowledge, e.g. the functionality of operators, which becomes a knowledge requirement. This is true for all steps involving selection or ordering as select operators, order operators, or sort suboperators.

6.1.2.2 Role properties

For the method-specific role properties we focus on the step refine, execute and repair operator in our knowledge model because it is the most knowledge demanding subtask—up to that level the subtasks consist mainly in heuristic search. In particular, the used knowledge concerns parameter initialisation, suboperator choice within a specialisation, result evaluation, and operator repair.

Initialisation knowledge The initial setting of operator parameters is in general performed by the method rule-based initialise parameters. This method uses the initialisation rule base of the operator to initialise all parameters without an established value. For this purpose the rule base must contain at least one rule for each parameter without value (*adequacy of*

the initialisation rule base to the unvalued parameters in the operator). Moreover, the rules serving to initialise a parameter should cover all possible situations in the application context of the rule base (*completeness of the rule set initialising a parameter*). Finally, there should not be two rules concluding different values for the same parameter in the same situation (*conflict-freeness of the rule set initialising a parameter*).

Choice knowledge In general the method rule-based choose suboperators is used to select the candidate(s) for a specialisation in a given situation. It uses the choice rule base attached to the operator decomposition for this purpose. The rule base should permit choosing every suboperator in the decomposition, i.e. it should contain at least one rule choosing each suboperator (*adequacy of the choice rule base to the suboperators in the specialisation*), and cover all possible situations (*completeness of the choice rule base*). Notice that there is no conflict in choosing two different suboperators in the same situation if the step specialise operator deals with this.

Evaluation knowledge The detection and diagnosis of possible problems after operator refinement or execution is usually carried out by the method rule-based evaluate results. Problems are signalled with an explicit indication to repair, and the corresponding diagnosis is expressed as a negative assessment on the operator application and/or tuning. The evaluation rule base of the operator should permit determining if a repair is necessary or not. For this purpose the rule base should cover all possible situations (*completeness of the evaluation rule base*). Additionally it should not derive conflicting conclusions, i.e. repair is necessary and repair is not necessary, in the same situation (*conflict-freeness of the evaluation rule base*).

Repair knowledge In general, a rule-based method is used to apply a corrective action to solve the problems diagnosed by rule-based evaluate results.

With a local repair, the repair rule base should treat all the problems diagnosed by the evaluation rule base (*adequacy of the repair rule base to the problems diagnosed by the evaluation rule base*).

With a non-local repair, the repair of $operator_j$ may be triggered by the repair rule base of $operator_i$ by means of a repair rule with an action `send-op(`$operator_j$, $problem_p$`)`. In this case, the repair rule base of $operator_j$ must also treat the diagnosis implicit in the transmission from $operator_i$, i.e. $problem_p$ (*adequacy of the repair rule base to the problems diagnosed by*

the evaluation rule base, and to the problems transmitted by any other operator). For instance a repair rule with the action `send-op(`$operator_j$, $problem_p$`)` in $operator_i$ needs at least one rule in the repair rule base of $operator_j$ treating the problem $problem_p$.

A non-local repair also requires that the overall repair path gives means to solve the problem, i.e. that the chaining of repair actions is continuous and ends with an action that can solve the problem (*repair path correctness*). For instance, if the action `send-op(`$operator_j$, $problem_p$`)` is included in the repair rule base of $operator_i$, in $operator_j$ there should be at least one repair rule to treat $problem_p$ with either a `re-execute` or a problem transmission to another operator which in turn complies with this. Besides, the chaining of repair actions must end and thus it is not advisable to transmit a problem to an operator that already takes part in the repair mechanism.

Additionally, the rules treating a diagnosed problem should cover all possible situations (*completeness of the rule set treating a problem*), and there should not be two rules concluding different repair actions in the same situation (*conflict-freeness of the rule set treating a problem*).

The adjustment of operator parameters is performed whenever the re-execution is required by the repair rule base of the operator. Therefore rule-based adjust parameters requires an adjustment rule base whenever the re-execution is used to repair the operator (*adequacy of the adjustment rule base to the treatments in the repair rule base*). In addition, this rule base should cover all situations in the application context (*completeness of the adjustment rule base*) and should not derive different values for the same parameter in the same situation (*conflict-freeness of the adjustment rule base*).

6.1.2.3 Desirable properties

Examples of knowledge characteristics more appropriate to the functioning of the analysed engines follow. They correspond to methods using different search strategies in performing their tasks.

In the search for solutions, PEG plan and execute only tries individual operators, under the assumption that they contain enough information to solve a problem. This means that PEGASE is suitable for knowledge bases containing hierarchical operators, and inversely, that non-hierarchical operator knowledge bases should not be used in combination with PEGASE. The exhaustive search of PUL plan and execute makes PULSAR more adequate to knowledge

bases containing non-hierarchical operators. Finally, the characteristics of the search for solutions in MedIA, make it suitable for managing hierarchical operators that need to be adapted at run-time.

The strategy in the search for a candidate in a specialisation decomposition has some implications on the characteristics of the choice criteria. In PUL specialise operator, the search in the set of candidates is exhaustive. An exhaustive search is more suitable for choice criteria selecting multiple suboperators since they are all considered as potential candidates.

6.2 Knowledge representation in LAMA

The implementation in LAMA of the PS concepts follows a hybrid knowledge representation scheme. In this scheme the knowledge concepts like operators or goals are represented by structured objects, whereas the different expert criteria are in general represented by rule bases. Figure 6.1 shows an example[1] of compound operator in the YAKL language[2]. The rule-based representation for criteria in LAMA is detailed next.

6.2.1 Rule-based representation

LAMA allows for the writing of rules using in preconditions mathematical relation expressions (with $=$, $<$, $>$, $<=$, $>=$, and $\neq$). The use of mathematical relations is limited to expressions having, on the left-hand side, a constant or a functional expression and, on the right-hand one, either a constant or an arithmetic expression (with $+$, $-$, $/$, and $*$). Constants represent specific elements of the knowledge base, e.g. constant values, operators, domain objects, arguments, parameters, or (sub)attributes of domain objects/arguments/parameters (e.g. $argument_l.attribute_m.attribute_n$). Variables can be declared and used within expressions as any other constant. Functional expressions often denote typical attributes of knowledge base elements, e.g. operator assessments have the form `assess-operator?(`$operator_j$`)`. There exist few basic functions, namely `type-of(`$argument_l$`)`, `assess-operator?(`$operator_j$`)`, `assess-parameter?(`$parameter_k$`)`, `assess-data?(`$argument_l$`)`, and `nb-previous-assess(`$operator_j$, $problem_p$`)`. Preconditions are related through the connective `AND`. Regarding rule actions, they can be either constant

[1] The examples in this chapter have been extracted from the Cerveau knowledge base [Thonnat et al., 1999].

[2] To increase readability, some YAKL keywords have been replaced with the terms employed in our knowledge modeling.

Compound Operator {	**name :**	segm-morpho-bin
...		
Functionality :	segmentation	
Characteristics :	math-morpho, binary	
Input Data		
	MedicalImage **name :**	mri-im
Input Parameters		
	symbol **name :**	computation-precision
	range :	[minimal, sharp, medium, coarse, maximal]
Output Data		
	MedicalImage **name :**	segmented-im
Preconditions		
	valid mri-im,	
	mri-im.physical-process = mri,	
	mri-im.physical-space = 3D	
	mri-im.intensity-coding = levels,	
	mri-im.form-situation = connected	
Postconditions		
	valid segmented-im,	
	segmented-im.intensity-coding = binary,	
	segmented-im.form-situation = alone	
Initialisation criteria		
...		
Evaluation criteria		
...		
Repair criteria		
...		
Decomposition		
	isolate-info-zone - extract-info - reconstruct-info	
...		
Distribution		
	segm-morpho-bin.mri-im → isolate-info-zone.init-im	
	reconstruct-info.complete-im → segm-morpho-bin.segmented-im	
Flow		
	isolate-info-zone.isolated-info-im → extract-info.connected-info-im	
	isolate-info-zone.isolated-info-im → reconstruct-info.reference-im	
	extract-info.extracted-info-im → reconstruct-info.incomplete-im	
}		

Figure 6.1: *Example of* LAMA *compound operator in* YAKL. The segm-morpho-bin operator serves to perform a binary segmentation on a 3D magnetic resonance image. It accepts as input a magnetic resonance image and produces a segmented one. The data type of the input and output images is the user-defined type MedicalImage, with attributes like physical-process, physical-space, intensity-coding and form-situation. The functionality of the segm-morpho-bin compound operator is achieved by the sequential application of the operators isolate-info-zone, extract-info and reconstruct-info.

Rule	**name**	init-computation-precision-1
	Let	?c a Context
	If	?c.required-precision = sharp
	Then	computation-precision := sharp
Rule	**name**	eval-segm-morpho-bin-1
	If	assess-data?(segmented-im) = regular-rounded-form
	Then	assess-operator good-quality continue
Rule	**name**	eval-segm-morpho-bin-2
	If	assess-data?(segmented-im) = irregular-form
	Then	assess-operator pb-medium-quality repair
Rule	**name**	rep-segm-morpho-bin-1
	If	assess-operator?(segm-morpho-bin) = pb-presence-connexions, nb-previous-assess(segm-morpho-bin, pb-presence-connexions) = 1
	Then	send-down reconstruct-info pb-too-many-details

Figure 6.2: *Examples of* LAMA *rules in* YAKL. All rules are attached to the segm-morpho-bin operator. The init-computation-precision-1 rule simply sets the parameter computation-precision to the value sharp if such is the required precision of the execution context (variable ?c of user-defined type Context). The rules eval-segm-morpho-bin-1 and eval-segm-morpho-bin-2 are evaluation rules indicating respectively continue and repair on the basis of the assessment of the argument segmented-im. Rule eval-segm-morpho-bin-2 diagnoses the problem simply as medium quality results. Finally, rep-segm-morpho-bin-1 repair rule indicates that if segm-morpho-bin operator has been diagnosed with pb-presence-connexions for the second time, it is likely that reconstruct-info suboperator is responsible due to an excess of details in its processing.

assignments the PS specific actions which have been described in chapter 5 (see page 49). Figure 6.2 shows examples of the different types of rules in YAKL.

Rules are grouped in rule bases with a particular role with respect to the element which they are attached to. This means that e.g. for the initialisation of the parameters of the segm-morpho-bin operator in figure 6.1, only the initialisation rules defined in the scope of this operator will be used (one of them appears in figure 6.2).

6.3 Knowledge base verification techniques

We have described in section 6.1 typical properties that a PS knowledge base should verify according to our knowledge model of different PS engines. These model-based properties are formulated using the terminology of the PS task and hence they are more significant than implementation-dependent ones.

The verification of the model-based properties needs different types of checks. Some properties only require referential checks between different objects in the knowledge base (many of those derived from the definitions in the PS domain model). Other properties require more complex structural checks involving several objects. Finally, rule base properties related to traditional rule-based anomalies require the application of specific techniques.

Referential checks The properties that can be verified by simple referential checks:

1. General, structural properties. We refer to syntactic-like properties derived from the concepts and relations in the PS domain model, and their semantics. For instance, compound operators must specify data distribution, and the arguments related through data distribution must be compatible.
2. Method-specific, structural properties. We refer to those properties corresponding to the knowledge requirements of the particular PS method. For instance, PEGASE requires that the functionality of operators is specified.

Structural checks The properties that require more complex structural checks:

3. General, structural properties:
   - Correctness of data distribution in specialisations.
   - Correctness of data distribution in sequences.
   - Correctness of data flow.
   - Correctness of hierarchical operators.
   - Completeness of the goal set with respect to the abstract functionalities in the operator knowledge base.
4. Method-specific, structural properties:
   - Correctness of data distribution in sequences (with optional operators).
5. Method-specific, role properties:
   - Adequacy of the initialisation rule base to the unvalued parameters in the operator.
   - Adequacy of the choice rule base to the suboperators in the specialisation.
   - Adequacy of the repair rule base to the problems diagnosed by the evaluation rule base, and to the problems transmitted by any other operator.

- Correctness of repair paths.
- Adequacy of the adjustment rule base to the treatments in the repair rule base.

6. Method-specific, desirable properties:
   - Adequacy of the operator knowledge base to the search strategy of plan and execute.
   - Adequacy of the choice rule base to the search strategy of specialise operator.

Rule base checks Finally, rule base properties requiring specific techniques:

7. General, structural properties:
   - Redundancy-freeness of rule bases.
8. Method-specific, role properties:
   - Completeness of rule bases:
     * Initialisation rule base: completeness of the rule set initialising a parameter.
     * Choice rule base: completeness of the rule base.
     * Evaluation rule base: completeness of the rule base.
     * Repair rule base: completeness of the rule set treating a problem.
     * Adjustment rule base: completeness of the rule base.
   - Conflict-freeness of rule bases:
     * Initialisation rule base: conflict-freeness of the rule set initialising a parameter.
     * Evaluation rule base: conflict-freeness of the rule base.
     * Repair rule base: conflict-freeness of the rule set treating a problem.
     * Adjustment rule base: conflict-freeness of the rule base.

The rule base properties above are related to rule-based anomalies that have been extensively studied in the field of V&V of knowledge-based systems. Next we present the rule-based techniques that we have chosen according to the LAMA representation.

6.3.1 Rule base verification techniques

Numerous techniques reviewed in chapter 2 involve verification of rule base implementation anomalies. They provide us with different definitions of consistency, redundancy and completeness anomalies, and their associated verification procedures. These techniques assume in general propositional logic or AOV-like rules.

For the consistency and redundancy checks there exist two main approaches, namely checks consistsing in pairwise rule comparisons and those considering all possible deductions, referred to as rule checks and rule extension checks respectively in [Preece and Shinghal, 1994]. Rule extension checks make no sense given the usual deduction models in PS rule-based methods, in which usually no rule chaining takes place as explained in chapter 5 (see pages 59 and 65). These deduction models are in tight relation with the rule-based representation in LAMA, in which rules conclude an action which directly constitutes an output. In our case rule checks are therefore sufficient. Various rule check algorithms can be found in [Suwa et al., 1982], [Puuronen, 1987] and [Cragun and Steudel, 1987].

For the completeness check, the usual approach consists in the exhaustive enumeration and examination of possible combinations of input variables. An exception to this approach is the logical definition for the completeness analysis of single-layered reactive rule bases in [Ligeza, 1997].

In preconditions of LAMA rules we usually find comparisons of constants or functional expressions against constant values (see figure 6.2). In order to apply definitions for AOV-like rules, this has been assumed for our implementations. Next we sketch the consistency, redundancy and completeness checks that we perform.

6.3.1.1 Consistency and redundancy checks

To check consistency and redundancy we use the notion of absolute logical uniqueness in [Cragun and Steudel, 1987]. Two rules have absolute logical uniqueness in their preconditions when a precondition making reference to an element (constant or functional expression) in one rule requires a different value in the other rule. This notion, which serves to ensure that two rules cannot fire simultaneously, is equivalent to the definition of mutually exclusive rules in [Pipard, 1988].

Then, if two rules have not mutually exclusive preconditions we perform additional checks to determine if their preconditions are exactly the same, or one subsumes the other. This helps us to detect redundant, subsumed and conflicting rules, depending on the results of the comparisons between rule actions.

6.3.1.2 Completeness check

If all the rules in the rule base are mutually exclusive, which means that there are not two rules that fire simultaneously, completeness can be checked numerically as in [Cragun and Steudel, 1987]. This also requires that preconditions only include equality comparisons. Otherwise completeness can be ensured by checking the non-existence of missing rules, by exhaustive enumeration of possible combinations of elements in preconditions.

Whenever completeness is not verified, bd-resolution [Ligeza, 1997] can serve to obtain an expression with respect to which the rule base is complete, which can help the expert to identify the missing rules. The basic form of bd-resolution is:

$$\frac{\psi_1 \wedge \omega,\ \psi_2 \wedge \neg\omega}{\psi_1 \wedge \psi_2}$$

When it is applied to the preconditions of two rules the obtained bd-resolvent describes the situations that they cover. For instance, if the two preconditions are $\psi_1 \wedge \omega$ and $\psi_2 \wedge \neg\omega$, in any situation satisfying $\psi_1 \wedge \psi_2$ either ω or $\neg\omega$ holds and thus one of the rules can be fired.

The extension of bd-resolution to the AOV formalism in [Ligeza, 1997] states that the condition to "glue" a set of formulas is that an element takes all its possible values. If the element e_i has as set of possible values $V_{e_i} = \{v_1, v_2, \ldots v_m\}$, this can be:

$$\frac{\psi_1 \wedge e_i = v_1,\ \psi_2 \wedge e_i = v_2,\ \ldots\ \psi_n \wedge e_i = v_m}{\psi_1 \wedge \psi_2 \wedge \ldots \psi_2}$$

Notice that these completeness checks need the definition of the set of possible values of precondition elements. This is specified in the case of domain objects/parameters/arguments by means of the corresponding range attributes. In the case of the assessments on operators/arguments/parameters, this is implicitly specified through the actions in the corresponding evaluation rule base. For instance, the rule eval-segm-morpho-bin-2 in figure 6.2 results in a new value pb-medium-quality in the domain of the assessment attribute of operator segm-morpho-bin.

6.4 Knowledge base verification within LAMA

The LAMA platform for PS system development has been presented as part of our research framework. Here we describe the issues for the integration of the necessary LAMA modules for the model-based verification PS knowledge bases.

The YAKL language for expertise definition is the interface that the knowledge engineer and the domain expert use for knowledge base development. The properties that require simple referential checks can be verified without a significant overload during the necessary parsing step of the input YAKL knowledge base into the LAMA constructs. This will allow the knowledge engineer and the domain expert to concentrate on knowledge elicitation early in the implementation stage. The verification of more complex properties, or more exhaustive verifications, can be performed once the knowledge base is deemed nearly finished, before operating with it in LAMA.

Therefore the referential checks will be performed by the LAMA YAKL parser. The verification of the rest of the properties will be performed by the LAMA knowledge base examiner, by using the structural information obtained from the parsing step. Figure 6.3 depicts these new LAMA modules and the way in which the knowledge engineer and the domain expert will interact with the platform during the development of a PS knowledge base.

Although the rationale of LAMA is to facilitate PS engine reconfiguration, no module serves to support this activity at present. For the development of a new engine, the designer implicitly or explicitly uses a semi-formal specification of the expert's reasoning process. This specification is then operationalised by combining BLOCKS instructions and structures in an algorithmic way. Once this LAMA engine has been validated with the help of the expert, the appropriate LAMA modules (the YAKL parser and the knowledge base examiner) have to be developed based on the characteristics of the PS method that the new LAMA engine implements.

In this light we have implemented a version of the YAKL parser and the knowledge base examiner to verify important knowledge requirements of both PEGASE and MedIA engines, which are used in the current applications in our team. For the implementation of the knowledge base examiner, we have emphasised the development of a verification library containing procedures to check typical model-based properties that PS knowledge bases have to verify. The idea behind this is to facilitate the configuration of future versions of the knowledge base examiner module adapted to the particular needs of new LAMA engines.

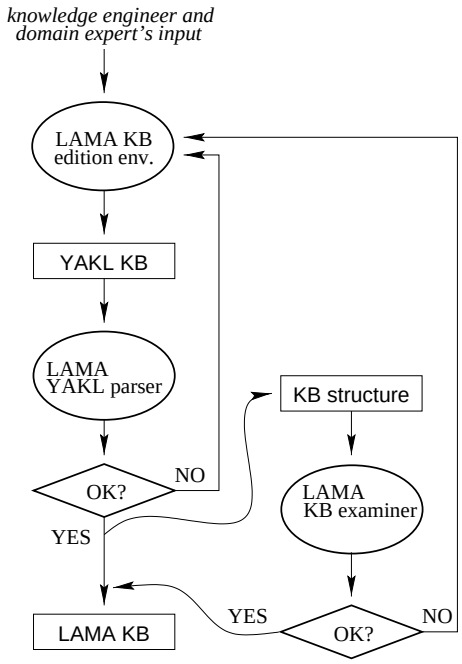


Figure 6.3: *The development of PS knowledge bases in the* LAMA *platform.* The knowledge engineer and the domain expert use a dedicated edition environment to enter the YAKL knowledge base. At any moment they can request the parsing of the evolving knowledge base, which will help them fix syntax errors or invalid references. Once the parsing step is successfully completed, the LAMA knowledge base examiner can be run to verify more complex properties.

6.5 Modules for knowledge base verification

6.5.1 YAKL parser

Besides YAKL syntax compliance, the YAKL parser performs many other important verifications. It verifies that the knowledge base complies with the general, structural properties that need simple referential checks. It also performs type checks, which are fundamental to ensure that object attributes are adequately referenced throughout the knowledge base.

The YAKL parser also verifies part of the general, structural properties requiring more complex structural checks, e.g. ensuring that an argument does not receive two values by means of data distribution. It even warns the expert about the gaps in the evolving knowledge base, e.g. highlighting that a parameter lacks initialisation means. These gaps, which will be pointed out as errors by the LAMA knowledge base examiner later on, are considered minor problems by the YAKL parser.

6.5.2 Knowledge base examiner

The module that has been implemented serves to verify the adequacy of the contents of a knowledge base to the most important knowledge requirements of PEGASE and MedIA engines. It is based on a verification library which at present contains procedures to check:

1. data distribution correctness in specialisation decompositions.
2. data distribution correctness in sequential decompositions (with optional operators).
3. data flow correctness.
4. hierarchical operator correctness.
5. adequacy of the initialisation rule base to the unvalued parameters in the operator.
6. adequacy of the choice rule base to the suboperators in the specialisation.
7. adequacy of the repair rule base to the problems diagnosed by the evaluation rule base.
8. repair path correctness.
9. adequacy of the adjustment rule base to the treatments in the repair rule base.
10. redundancy- and conflict-freeness of rule bases.
11. completeness of rule bases.

The basic procedures for rule base verification are applied differently according to the role of the rule base. For example, initialisation rule bases are split into as many rule sets as parameters they initialise, which are verified independently.

The algorithms of these procedures can be found in appendix C. Next we detail the procedures that perform the rule base checks.

6.5.2.1 Redundancy and conflict freeness of rule bases

A description of the algorithm to check a rule base for redundancy and conflicts comes next (see details in appendix C). Every rule is compared against every other rule. If their preconditions have absolute logical uniqueness they are marked as exclusive, which means that they do not need further checks. If preconditions have no absolute logical uniqueness and all refer to the same elements (constants or functional expressions), which means that all elements appear in both rules making reference to the same value, they are marked as equivalent. Else, if preconditions have no absolute logical uniqueness and all the elements in the preconditions of one rule are included in the preconditions of the other, they are marked as subsumed, which means that the former fires whenever the latter does it; otherwise as non exclusive.

A pair of rules with exclusive preconditions will never fire simultaneously and therefore they are marked as **mutually exclusive rules**. This is important for later completeness checks. Before checking a rule base for redundancy and conflicts all rules are considered non mutually exclusive, then any absolute logical uniqueness is looked for to show the contrary.

Redundant, subsumed and conflicting rules. Rules with equivalent preconditions always fire simultaneously whereas rules with subsumed preconditions will fire simultaneously in some situations. They are all further studied by determining whether their actions are equivalent, conflicting or merely different. Rule actions are deemed conflicting according to their semantics.

Then, two rules are considered redundant when both their preconditions and actions are equivalent. The rules are considered subsumed when their preconditions are subsumed and their actions are equivalent. Finally, the rules are considered conflicting if their preconditions are equivalent or subsumed and their actions are conflicting.

For example, in the rule base fragment in figure 6.4, the rules init-computation-precision-1 and init-computation-precision-2 do not have absolute logical uniqueness. As the precondi-

Rule	**name**	init-computation-precision-1
	Let	?c a Context
	If	?c.required-precision = sharp
	Then	computation-precision := sharp
Rule	**name**	init-computation-precision-2
	Let	?c a Context
	If	?c.required-precision = sharp,
		mri-im.form-connexions = weak
	Then	computation-precision := minimal
Rule	**name**	init-computation-precision-3
	Let	?c a Context
	If	?c.required-precision = sharp
		mri-im.spot-presence $\neq$ no
	Then	computation-precision := minimal

Figure 6.4: *Subset of initialisation rule base of operator segm-morpho-bin*

tions of init-computation-precision-1 are included in the preconditions of init-computation-precision-2, they are marked as subsumed. Finally, since initialisation actions assigning different values are in conflict, the rule pair is considered as conflicting.

Finally, there exist rules which preconditions do not have absolute logical uniqueness, are not equivalent nor subsumed. These rules remain as **non mutually exclusive rules**. They might fire simultaneously, and therefore the expert should be aware of their potential interactions. In consequence the preconditions of the two rules are used to describe the situations in which they can both fire. This expression can help the expert to determine the correct form of the rules if necessary.

For example, the rules init-computation-precision-2 and init-computation-precision-3 in figure 6.4 are non mutually exclusive. Then the expression describing the situation in which they both fire (?c.required-precision = sharp $\wedge$ mri-im.form-connexions = weak $\wedge$ mri-im.spot-presence $\neq$ no) will be presented to the expert.

6.5.2.2 Completeness of rule bases

A description of the different algorithms to check a rule base for completeness follows (see details in appendix C).

Numerical completeness check If all rules in the rule base are mutually exclusive and provided that preconditions only include equality comparisons, completeness can be checked

Rule	**name**	init-parameters-1
	If	amplitude = minimal
	Then	iterations := 1
Rule	**name**	init-parameters-2
	If	amplitude = sharp, isotropy = yes
	Then	iterations := 2
Rule	**name**	init-parameters-3
	If	amplitude = sharp, isotropy = quasi
	Then	iterations := 1

Figure 6.5: *Subset of initialisation rule base of operator dilatation*

by summing the disjoint cases covered over the different rules, and comparing the result against the number of combinations of possible values of precondition elements.

For example, given the fragment of rule base in figure 6.5, where the sets of possible values of amplitude and isotropy are respectively $V_{amplitude} = \{minimal, sharp, medium, coarse, maximal\}$ and $V_{isotropy} = \{yes, quasi, no\}$, it can be seen that rule init-parameters-1 covers three cases, and init-parameters-2 and init-parameters-3 cover one case each. Notice that init-parameters-1 does not indicate any value for isotropy and thus covers as many cases as different values isotropy may take, i.e. $card(V_{isotropy})$. Since the number of combinations $card(V_{amplitude}) \cdot card(V_{isotropy})$ is 15, which is greater than the 5 cases covered, we can say that the rule base fragment is not complete.

Enumeration completeness check When the rule base contains redundancy or ambiguity, or precondition relations other than equality, completeness has to be checked by other means. Completeness check by enumeration consists in trying to find a fireable rule for every combination of values of precondition elements. In general not all value combinations are to be tried. For instance, in the rule base in figure 6.5, once init-parameters-1 rule has been identified as fireable with $\{amplitude = minimal\}$ no more combinations with this value are tried (e.g. $\{amplitude = minimal, isotropy = yes\}$).

Completeness characterisation with bd-resolution To obtain an expression with respect to which the rule base is complete, we search for a bd-resolvent for the preconditions of every rule pair. Inconsistent bd-resolvents are rejected, e.g. including $e_j = v_1$ and $e_j \neq v_1$. If

Rule	**name**	adj-computation-precision-1
	If	assess-parameter?(computation-precision) = too-sharp, nb-previous-assess(reconstruct-info, pb-missing-details) = 1
	Then	computation-precision +=
Rule	**name**	adj-computation-precision-2
	If	assess-parameter?(computation-precision) = too-coarse
	Then	computation-precision −=

Figure 6.6: *Adjustment rule base of operator reconstruct-info.* The rule adj-computation-precision-2 is fireable when the assessment of the parameter computation-precision is too-coarse, whichever the value of nb-previous-assess(reconstruct-info, pb-missing-details) is.

a bd-resolvent is found for the preconditions of two rules, it will be kept. The process ends when all possible rule pairs have been explored. Then, the disjunction of the obtained bd-resolvents and of the preconditions of rules for which no resolvent was found will provide an expression with respect to which the rule base is complete. A simplification of this expression is performed in order to obtain a more concise form to characterise rule base completeness.

We also take into account that a rule that does not indicate any value for an attribute is shorthand for different rules. For instance, the incomplete rule base in figure 6.6 can be unfolded into the one in figure 6.7. Then, considering that the possible assessments of computation-precision are too-sharp and too-coarse, we proceed as follows. The preconditions of adj-computation-precision-1 and adj-computation-precision-21 can be "glued" on the basis of assess-parameter?(computation-precision), resulting in the bd-resolvent (nb-previous-assess(reconstruct-info, pb-missing-details) = 1). The same happens with adj-computation-precision-21 and adj-computation-precision-22, resulting in (assess-parameter?(computation-precision) = too-coarse). The obtained expression is:

$$((\text{nb-previous-assess(reconstruct-info, pb-missing-details)} = 1) \\ \vee (\text{assess-parameter?(computation-precision)} = \text{too-coarse}))$$

which means that, e.g., the following situation is not covered:

$$((\text{nb-previous-assess(reconstruct-info, pb-missing-details)} \neq 1) \\ \wedge (\text{assess-parameter?(computation-precision)} = \text{too-sharp}))$$

Rule	**name**	adj-computation-precision-1
	If	assess-parameter?(computation-precision) = too-sharp,
		nb-previous-assess(reconstruct-info, pb-missing-details) = 1
	Then	computation-precision +=
Rule	**name**	adj-computation-precision-21
	If	assess-parameter?(computation-precision) = too-coarse,
		nb-previous-assess(reconstruct-info, pb-missing-details) = 1
	Then	computation-precision −=
Rule	**name**	adj-computation-precision-22
	If	assess-parameter?(computation-precision) = too-coarse,
		nb-previous-assess(reconstruct-info, pb-missing-details) ≠ 1
	Then	computation-precision −=

Figure 6.7: *Adjustment rule base of operator reconstruct-info after unfolding.*

6.6 Application of the modules for knowledge base verification

The LAMA YAKL parser has been developed using bison[3]. The different verifications that it performs are implemented in C++, as semantical actions attached to syntax rules. The LAMA knowledge base examiner has been implemented in GNU C++. It is based on a GNU C++ library implementing the different knowledge base verification procedures that we have described in subsection 6.5.2.

The implemented YAKL parser and knowledge base examiner have been tested with two knowledge bases for the supervision of image processing programs in different application domains: Cerveau and Progal. Both have been intended to work with PEGASE engine.

Cerveau [Thonnat et al., 1999] is a knowledge base to perform brain segmentation on 3D magnetic resonance images. Cerveau consists of 37 operators, where 16 of them are compound ones. It contains 57 rule bases of a size varying from 1 to 16 rules.

Progal knowledge base [Thonnat et al., 1999] has as objective the description of the morphology of galaxy images in terms of pertinent numerical parameters. Progal consists of 92 operators, where 30 of them are compound ones. It contains 40 relatively small rule bases (of a size from 1 to 6 rules).

In the development of these knowledge bases, the simple verifications performed by the YAKL parser have been shown to be very useful early in the implementation. Given the complex organisation of PS knowledge bases, domain experts are prone to make mistakes and

[3]Bison is the GNU Project parser generator (yacc replacement).

therefore the checks that the parser performs are crucial. In addition to this, the verifications carried out by the LAMA knowledge base examiner are essential. Most of the properties that are checked can hardly be verified by hand, due to the complex verifications that they need, and could go unnoticed. Among the most frequent errors detected in Cerveau and Progal by the LAMA knowledge base examiner we can cite:

- data distribution incorrectness.
- inadequacy of the initialisation rule base to the unvalued parameters in the operator: a parameter lacks initialisation means.
- inadequacy of the adjustment rule base to the treatments in the repair rule base: either the re-execution is required in the repair rule base and no adjustment rule base exists or, inversely, an adjustment rule base exists and the re-execution is not required.
- inadequacy of the repair rule base to the problems diagnosed by the evaluation rule base: e.g. the evaluation rule base is able to diagnose problems but there is not a repair rule base to solve them.
- repair path incorrectness.
- rule base inconsistency, usually derived from rules with subsumed preconditions.
- rule base incompleteness.

Our experiences in the verification of Cerveau and Progal knowledge bases have confirmed that errors occur very often [Marcos et al., 1998b]. Many of them concern data distribution and repair paths, which is not surprising since the specification of the necessary knowledge is a difficult task. For instance, the design of a correct repair mechanism implies keeping track of different repair paths across several operators. In addition, numerous errors are related to the inadequacy of the contents of rule bases to their intended role. Finally, many rule-based anomalies occur despite the small size of rule bases.

6.7 Conclusions

The model-based verification properties that we have defined in this chapter correspond to typical properties that a PS knowledge base should verify according to the requirements of different engines. These model-based properties are more significant than the usual implementation-dependent ones. They can be exploited not only for the verification of implemented knowledge bases but also prior to the implementation stage.

We have exploited the model-based verification properties described above in the design and implementation of the necessary knowledge base verification modules. The YAKL parser and the LAMA knowledge base examiner implemented in this light have shown to be a valuable help in the development of correct PS knowledge bases. What is more, they serve to verify the adequacy of the knowledge base contents to the main knowledge requirements of PEGASE and MedIA engines.

The YAKL parser and the LAMA knowledge base examiner carry out different types of checks. Regarding the rule base checks, the simple techniques that we have employed, i.e. rule checks for consistency and redundancy verification and enumeration check for completeness verification, seem to be sufficient for PS knowledge bases. A negative aspect to mention is related to our algorithm for completeness characterisation by bd-resolution. In general it fails to find a concise expression due to the pairwise rule comparisons on which it is based. With the exception of few cases, as the example from Cerveau in page 97, in general it results in a useless expression. An improvement in this direction is necessary in order to use bd-resolution for the characterisation or even the check of completeness.

Finally, in addition to the implemented verification modules useful for PEGASE and MedIA knowledge bases, we have sought the development of a verification library with procedures to check typical model-based properties. We expect that these verification procedures will be useful in the implementation of future verification modules in the framework of the LAMA platform.

Chapter 7

Verification and Validation of Program Supervision Engines

The verification and validation of inference engines comes to relevance in frameworks supporting the (re)configuration of engines, as is the case of Lama. Inference engines are software and as such the application of software engineering verification techniques is necessary to ensure their reliability. With the purpose of assessing the feasibility of the use of these techniques in the development of PS systems, we have carried out some experiments directed towards the V&V of PS engines.

The structure of this chapter is the following. We first detail the objectives of our experiments on the V&V of PS engines. Then we introduce KIV, which is the verification tool that we have used, and justify its adequacy to the planned objectives. Afterwards we describe our experiences and finish with some conclusions on the use of KIV and the V&V of PS engines.

7.1 Experimental objectives

The application of software verification techniques in the development of PS engines is fundamental to ensure a certain degree of reliability. In Lama, PS engines are implemented by combining Blocks instructions in an algorithmic way. There are different V&V concerns in connection with the reliability of PS engines constructed in this way. First, the implementation of the instructions in the Blocks library must be reliable, i.e. it must be ensured

that they terminate and that they correspond to their specified competence. Second, when configuring a PS engine from BLOCKS instructions, it must be assured that the algorithmic composition of instructions is correct, and that it is adequate to solve the PS task. The former regards the V&V of BLOCKS instructions and the latter is the V&V of compositions of instructions.

The V&V of BLOCKS instructions is unavoidable because they are reused several times and this increases the cost of errors [Fensel et al., 1998]. Also for reusability reasons these instructions have a well-defined competence. In some cases this competence is close to their implementation. Such is the case of the BLOCKS counterparts of the steps execute-operator, test-preconditions or apply-effects in our knowledge model. Considering these characteristics, we have chosen to focus our experiments on the V&V of an instruction implemented from elementary instructions as the previously mentioned, which are assumed to be reliable. This has allowed us to assess the feasibility of the use of software engineering verification techniques for the V&V of PS engines, and also as a formal means for the detection of assumptions.

7.2 KIV

KIV (*Karlsruhe Interactive Verifier*) [Reif, 1995] is an advanced tool for the verification of modular software systems. It has been applied to the verification of different applications up to a size of several thousand lines [Stenzel, 1993], [Fuchß 1994], [Fuchß et al., 1995]. KIV supports the entire development process, that is, the specification, the implementation and the verification of software systems.

Specification For the specification, descriptions in the style of abstract data types [Ehrig and Mahr, 1985] are used. They are based on signatures with sort and operation symbols, on variable declarations, and on a set of first-order axioms describing the sorts and operations.

Structuring operations can be used to break a specification down into more tractable components. Examples of structuring operations are union, enrichment, and actualization of specifications. A union specification, $SPEC_1 + SPEC_2$, describes the union of the elements in $SPEC_1$ and $SPEC_2$. An enrichment specification, **enrich** $SPEC$ **with** Δ, is used to add the new elements in Δ to the specification $SPEC$, i.e. new sorts and operations, and new axioms describing them. Parameterized specifications serve to define generic data types, e.g.

string of data, and include for this purpose a formal parameter specification, e.g. data. A specification **actualize** $SPEC_1$ **with** $SPEC_2$ **by morphism** h actualizes a parameterized specification $SPEC_1$, replacing its formal parameter by the actual specification $SPEC_2$. It includes a morphism h describing the renamings and/or identifications of sort and operation symbols.

Implementation The different specifications within a system can be refined or implemented by independent modules. A module implements the operations in an export specification on the basis of the elements in an import specification. It therefore consists of the export specification, the import specification, and the implementation defining the collection of procedures for the operations in the export specification. Procedures are written in a Pascal-like syntax. Since different specifications are refined separately, KIV enforces the modularisation of software designs.

Verification The correctness of modular systems in KIV is reduced to the correctness of single modules by imposing restrictions on the refinement of specifications into modules (see [Reif, 1995] for details). The proof obligations that are necessary and sufficient for the correctness of single modules are generated automatically by KIV. These proof obligations, expressed as sequents in dynamic logic[1], ensure that modules terminate and that they terminate in a state that complies with the axioms in the export specification. The next step consists in actually proving these obligations.

For the verification, KIV theorem prover offers a variety of proof engineering facilities that eases the organisation of proof sessions. Particularly, proof trees are visualised and can be manipulated (browsed, pruned, replayed, etc.) with the help of a graphical user interface. Besides, KIV implements a number of heuristics that automate proofs to a large extent. Nevertheless, as complex proofs cannot in general be automated, KIV integrates automated reasoning and user's proof expertise. The user constructs proofs interactively by selecting a proof step (e.g. symbolic execution) from the menu that is proposed by the system. Other examples of user interaction are the input of a lemma or an induction hypothesis, the selection of different heuristics when the system gets stuck, or the decision to backtrack.

[1] Dynamic logic extends ordinary predicate logic by formulas $[\pi]\varphi$ and $\langle\pi\rangle\varphi$, where π is a program and φ is again a dynamic logic formula. The intuitive meaning of $[\pi]\varphi$ is: "if π terminates, φ holds after the execution of π". The formula $\langle\pi\rangle\varphi$ is understood as: "π terminates and φ holds after the execution of π". A program π is a pair $(decl \mid \alpha)$, i.e. a declaration plus a command α. If the declaration can be understood from the context, it will be omitted. A dynamic logic sequent $\Gamma \vdash \Delta$, where Γ and Δ are lists of formulas, holds if the conjunction of the formulas of Γ implies the disjunction of the formulas of Δ.

7.3 Adequacy of KIV to our experimental objectives

We target the specification and verification of big-grain sized instructions implemented on the basis of the specification of simple BLOCKS instructions. This verification is closely related to the detection of the hidden assumptions that are made in the implementation.

KIV allows the structuring of specifications and fosters the modularisation of software. The structuring of specifications makes it suitable for the specification of both the competence of BLOCKS instructions and of algorithmic compositions of these, based on the former. On the other hand, software modularisation allows the reduction of the overall V&V effort thanks to the principle on modular system correctness. This is fundamental when facing the V&V of PS engines.

In addition, the interactive nature of proofs with KIV makes it also suitable for searching for assumptions by analysis of failed proofs. This has been called inverse verification in [Fensel and Schönegge, 1998]. KIV returns an open goal when it fails to finish a proof, which can be used as the starting point in the process of constructing assumptions.

Finally, KIV uses dynamic logic, which has been proven to be useful in the specification of knowledge-based systems [Fensel et al., 1998], [Fensel and Schönegge, 1998]. The advantages of dynamic logic are that it is quite expressive and that programs take part in formulas, resulting in more readable formulas and proofs.

7.4 Verification and validation of BLOCKS-based instructions

Our aim is to assess the feasibility of the utilisation of KIV for the specification and verification of instructions implemented on the basis of the specification of simple BLOCKS instructions, and for the detection of the assumptions that are made in their implementation. Next we describe the steps that we follow in our experiment.

We will first specify the competence of the necessary BLOCKS instructions. As they are assumed to be reliable, they will not be further refined. This requires the specification of the elementary data types used, e.g. the operator data type. The specifications of these BLOCKS instructions will then be used via the KIV structuring operations (e.g. sum or enrichment) for the specification of the bigger instruction which we target at (**specification**). All instructions will be specified functionally by using axioms that describe their competence independently of how they are implemented.

Then, the specification of the target instruction will be refined by a module implementing it on the basis of the specifications of the necessary BLOCKS instructions (**implementation**). The procedures to implement the required instruction will approximately reflect the usual control in PS engines, except for the details.

Finally, the proof of the module correctness (**verification**) will ensure the termination of the implementation and the competence as defined in the specification, and might serve to identify the assumptions necessary to provide this competence, if there were any.

Our work is inspired by the KIV utilisation in [Fensel and Schönegge, 1997] and [Fensel et al., 1998]. KIV is used in these references in the verification of different proof obligations necessary to relate the elements in the specification of a knowledge-based system according to the framework described in [Fensel et al., 1996]. Given a task definition and the description of a problem-solving method to solve it, KIV is used to prove the termination and correctness of the operational specification of the problem-solving method, and to prove that its competence can solve the defined task. The latter serves to detect the assumptions that are necessary to close the gap between the competence of the problem-solving method and the requirements of the task. We target the verification of the termination and correctness of algorithmic compositions of BLOCKS instructions, which is in the line of the former proofs.

7.5 Verification and validation of the decompose-sequence instruction

As a first experience we have specified, implemented and verified a simplified version of the decompose-sequence step in our knowledge model. This step performs the hierarchical refinement or the execution of the suboperators within a sequential decomposition. In order to reduce the complexity of the implementation, and consequently the difficulty of proofs, we have considered the following simplifications:

- the elementary data types, i.e. state and operator, have been simplified. For instance, the operator data type comprises only sequential decompositions and does not include initialisation, evaluation nor repair knowledge.
- in relation to the decomposition characteristics, the hierarchical refinement is limited to sequential decompositions.

- in connection with the absence of initialisation, evaluation and repair knowledge, the hierarchical refinement and execution steps are performed without parameter initialisation and evaluation-repair loop.

These simplifications have allowed us to focus on the verification of the hierarchical refinement of sequential decompositions and the structural properties that it may require, which we have deemed convenient as first experience. The instructions for parameter initialisation, result evaluation and operator repair could have been specified as we do with simple BLOCKS instructions, without any implementation, and used as such after the hierarchical refinement and execution steps. However, this would not have significantly contributed to the termination of the hierarchical refinement or the structural properties that it assumes.

The development graph in figure 7.1 shows how we have specified and implemented in KIV the decompose-sequence instruction. It consists of the specification of the competence of this instruction (*decompose-sequence-competence*), the module *decompose-sequence-module* implementing it, and the numerous specifications on which this implementation is based. The most important ones are those corresponding to the state and operator data types (*state* and *operator*), and to the competence of the execute-operator basic instruction (*execute-operator-competence*). Part of the elements in the KIV library of specifications, which includes specifications and verified implementations of common data types, has been used (e.g. *set*). A complete listing of the specifications in figure 7.1 can be found in appendix D.

Next we introduce the specifications *state*, *operator*, *execute-operator-competence* and *decompose-sequence-competence*. Then we present the module *decompose-sequence-module* and finally the proofs that we have carried out in KIV to ensure its correctness.

7.5.1 Specification

State specification Regarding the state data type, we have considered that its contents are organised as unique identifiers with an associated value. The *valued-item* specification in figure 7.2 introduces the sort *vitem* describing pairs identifier-value. Such pairs are generated by the constructor *mkvitem*. The selectors *.vitid* and *.vitval* are postfix functions returning the corresponding arguments of an object of sort *vitem*.

In KIV it is possible to define functions and predicates to be used as postfix or prefix if they have one parameter, and as infix if they have two. This is indicated by a dot at one or

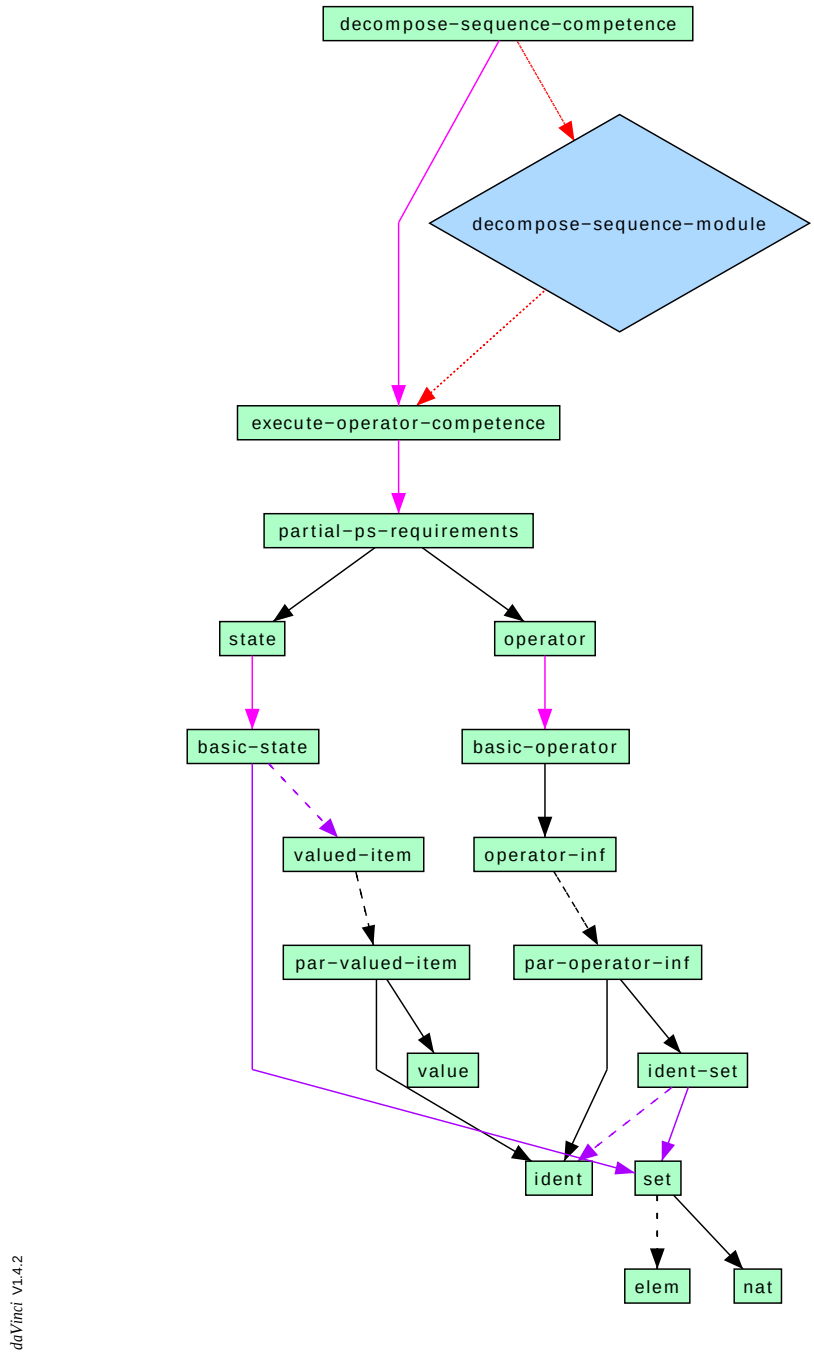


Figure 7.1: *Development graph of the* decompose-sequence *instruction.* Specifications are represented by rectangles and modules by diamonds. Various types of arrows indicate the different dependences between elements. For instance, the module `decompose-sequence-module` has the import `execute-operator-competence` and the export `decompose-sequence-competence`.

valued-item =
generic data specification
    parameter par-valued-item
    vitem = mkvitem (. .vitid : ident, . .vitval : value);
    variables vi: vitem;
end generic data specification

Figure 7.2: *The* `valued-item` *specification*

basic-state **actualize** set **with** valued-item **by morphism**
  set → state, elem → vitem, @ → $@_{st}$, insert → insertst, \ → $\setminus_{st}$, select → selectst,
  # → $\#_{st}$, ∈ → $\in_{st}$, ⊂ → $\subset_{st}$, s → st, s' → st', s_0 → st_0, s_1 → st_1, s_2 → st_2, e
  → vi, e_0 → vi_0, e_1 → vi_1, e_2 → vi_2
end actualize

Figure 7.3: *The* `basic-state` *specification*

both sides of the operation symbol: $.func\_or\_pred$ for a postfix operation, $func\_or\_pred.$ for prefix, and $.func\_or\_pred.$ for infix.

The *basic-state* specification in figure 7.3 defines a sort for sets of *vitem* objects as an actualization of the KIV *set* specification with *valued-item*. The morphism identifies, among other elements, the empty state as $@_{st}$, the function inserting an element in a state as *insertst*, and the membership relation as $\in_{st}$.

Finally, the *state* specification in figure 7.4 enriches *basic-state* by introducing the predicate *is-item-in-state*. This predicate, which determines whether a given identifier appears in the identifier-value pairs in a state, has been used to define the competence of the decompose-sequence and execute-operator instructions. The specification also includes an axiom to ensure the uniqueness of identifiers in a state, and two others to describe the new predicate.

Operator specification Regarding the operator data type, input and output arguments are specified as sets of identifiers, likewise parameters. As we have referred to before, only sequential decompositions are considered for compound operators. Within decompositions, no specification of data distribution nor flow is handled. It is assumed instead that the connections between operator arguments/parameters and suboperator ones are established on the basis of their identifiers. The specification of the operator data type is split over several ones.

The *ident-set* specification in figure 7.5 defines the sort *identset* for sets of identifiers by means of an actualization of *set* with the basic specification *ident*. Operator arguments and

state =
enrich basic-state **with**
predicates is-item-in-state : state × ident;
variables st: state;

axioms

$\forall$ vi, vi_0.vi $\in_{st}$ st $\wedge$ vi_0 $\in_{st}$ st $\wedge$ vi $\neq$ vi_0 $\rightarrow$ vi.vitid $\neq$ vi_0.vitid,
$\neg$ is-item-in-state($@_{st}$, id_0),
is-item-in-state(insertst(vi_0, st), id_0)
$\leftrightarrow$ id_0 = vi_0.vitid $\vee$ is-item-in-state(st, id_0)

end enrich

Figure 7.4: *The* `state` *specification*

ident-set **actualize** set **with** ident **by morphism**
set $\rightarrow$ identset, elem $\rightarrow$ ident, @ $\rightarrow$ $@_{ids}$, insert $\rightarrow$ insertids, $\setminus$ $\rightarrow$ $\setminus_{ids}$, select $\rightarrow$ selectids, # $\rightarrow$ $\#_{ids}$, $\in$ $\rightarrow$ $\in_{ids}$, $\subset$ $\rightarrow$ $\subset_{ids}$, s $\rightarrow$ ids, s' $\rightarrow$ ids', s_0 $\rightarrow$ ids_0, s_1 $\rightarrow$ ids_1, s_2 $\rightarrow$ ids_2, e $\rightarrow$ id, e_0 $\rightarrow$ id_0, e_1 $\rightarrow$ id_1, e_2 $\rightarrow$ id_2
end actualize

Figure 7.5: *The* `ident-set` *specification*

parameters are specified on the basis of this sort.

The information common to primitive and compound operators, i.e. identifier, functionality, set of input and output arguments, and set of parameters, is defined in the sort ***opinf*** of the ***operator-inf*** specification (see figure 7.6). The sort ***opinf*** is generated by the constructor ***mkopinf***. The selectors ***.operid***, ***.operfunc***, ***.operinput***, ***.operoutput*** and ***.operparam*** return the corresponding arguments of an ***opinf***.

The sorts ***operator*** and ***decomposition***, which are recursively defined one in terms of the other, are presented in the ***basic-operator*** specification in figure 7.7. These sorts allow to construct compound operators with arbitrary levels of abstraction as the example in figure 7.8

operator-inf =
generic data specification
parameter par-operator-inf
opinf = mkopinf (. .operid : ident, . .operfunc : ident,
. .operinput : identset, . .operoutput : identset, . .operparam : identset);
variables opi: opinf;
end generic data specification

Figure 7.6: *The* `operator-inf` *specification*

```
basic-operator =
data specification
    using operator-inf
    operator = mkprimop (. .priminf : opinf) with isprimop
             | mkcompop (. .compinf : opinf, . .dec : decomposition) with iscompop
             ;
    decomposition = nildec
                  | . seq . (carseq : operator, cdrseq : decomposition) with isdecseq
                  ;
    variables op: operator; dec: decomposition;
end data specification
```

Figure 7.7: *The* `basic-operator` *specification*

shows. In addition to constructors and the corresponding selectors, the specification includes the predicates *isprimop*, *iscompop* and *isdecseq*. The first two predicates respectively test whether an operator has been generated by *mkprimop* or *mkcompop*. Similarly, *isdecseq* tests whether a decomposition has been generated by the application of *seq*.

Finally, the *operator* specification in figure 7.9 enriches the sorts *operator* and *decomposition* with the predicate *indec* and the axioms that describe it. This predicate, which serves to determine whether an operator, primitive or compound, appears within a decomposition, has been used in proofs by structural induction on operator.

Specification of execute-operator **competence** The execute-operator step basically calls the program associated to the primitive operator on which it is applied, modifying the current state with the results of program execution. The *partial-ps-requirements* specification, which merely consists in the union of *state* and *operator* ones, is the basis for the specification of the competence of execute-operator.

The specification *execute-operator-competence* in figure 7.10 introduces the function *execute-operator* and defines its competence. The sole competence that can be attributed is that it changes the value of operator outputs if they already exist in the state, or generates a new value otherwise. Again, in order to reduce the complexity of proofs, we have chosen to simplify this competence. Then, the first axiom simply states that after the execution of a primitive operator all its outputs appear in the final state. The second axiom, which states that all the elements in the initial state persist in the final state after operator execution, has been added in the course of verification.

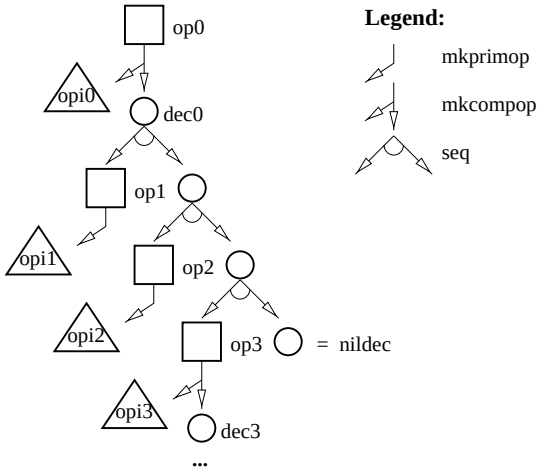


Figure 7.8: *Example of compound operator construction.* A compound operator op0 describing a sequence of two primitive operators (op1, op2) plus a compound one (op3), is constructed through the expression mkcompop(opi0, mkprimop(opi1) seq mkprimop(opi2) seq mkcompop(opi3,dec3) seq nildec).

operator =
enrich basic-operator **with**
predicates indec : operator × decomposition;

axioms

¬ indec(op, nildec),
isprimop(op_0)
→ (indec(op, op_0 seq dec) ↔ op = op_0 ∨ indec(op, dec)),
iscompop(op_0)
→ (indec(op, op_0 seq dec) ↔ op = op_0 ∨ indec(op, op_0.dec) ∨ indec(op, dec))

end enrich

Figure 7.9: *The* `operator` *specification*

execute-operator-competence =
enrich partial-ps-requirements **with**
functions execute-operator : operator × state → state ;
variables primitive: operator; ini-state: state;

axioms

isprimop(primitive)
→ (∀ id. id $\in_{ids}$ primitive.priminf.operoutput
→ is-item-in-state(execute-operator(primitive, ini-state), id)),
isprimop(primitive)
→ (∀ id. is-item-in-state(ini-state, id)
→ is-item-in-state(execute-operator(primitive, ini-state), id))

end enrich

Figure 7.10: *The* `execute-operator-competence` *specification*

decompose-sequence-competence =
enrich execute-operator-competence **with**
functions decompose-sequence : operator × state → state ;
variables sequence: operator; fin-state, ini-state: state;

axioms

iscompop(sequence)
→ (decompose-sequence(sequence, ini-state) = fin-state
→ (∀ id.id $\in_{ids}$ sequence.compinf.operoutput → is-item-in-state(fin-state, id)))

end enrich

Figure 7.11: *The* `decompose-sequence-competence` *specification*

Specification of decompose-sequence **competence** The specification in figure 7.11 describes the decompose-sequence instruction. It enriches `execute-operator-competence` by introducing the function `decompose-sequence` and defining its competence. The only axiom states that after the decomposition of a compound operator all its outputs appear in the final state, analogously to `execute-operator`.

From the data specifications, KIV automatically generates a set of axioms which include:

- Test predicates for constructors, e.g. :
  ⊢ isprimop(mkprimop(opi))
- Injectivity of constructors, e.g. :
  ⊢ mkprimop(opi) = mkprimop(opi0) ↔ opi = opi0

- Uniqueness of constructors, e.g. :
  $\vdash$ mkprimop(opi) $\neq$ mkcompop(opi0,dec)
- Case distinction of constructors, e.g. :
  $\vdash$ op = mkprimop(op .priminf) $\vee$ op = mkcompop(op .compinf, op .dec)

7.5.2 Implementation

The module *decompose-sequence-module* implements the function *decompose-sequence* in the export specification *decompose-sequence-competence* on the basis of the elements in the import specification *execute-operator-competence*. For this purpose it provides the procedure *decompose-sequence#*, which accepts an operator and an input state, and directly returns the input state if it is called with a primitive operator; otherwise, it calls the procedure *control#*. The latter is a recursive procedure that accepts a decomposition and an input state and returns the state resulting from the successive decompositions or executions of the operators in the sequence:

```
decompose-sequence =
module
    export decompose-sequence-competence
    refinement
      representation of operations
          decompose-sequence#   implements   decompose-sequence;

      import execute-operator-competence

      procedures control#(decomposition, state) : state;

      variables out: state;

      implementation

     decompose-sequence#(op, st0; var out)
     begin
       if isprimop(op) then out := st0 else var dec = op.dec in
       control#(dec, st0;out)
     end
```

```
control#(dec, st0; var out)
begin
   if dec = nildec then out := st0 else
     var op = carseq(dec), st = @st in
     begin
        if isprimop(op) then st := execute-operator(op, st0) else control#(op.dec, st0;st);
        control#(cdrseq(dec), st;out)
     end
end
```

7.5.3 Verification

From the previous implementation and the export specification *decompose-sequence-competence* that it refines, KIV automatically generates the following proof obligations:

i-1 Termination of *decompose-sequence#*:

$\vdash$ ⟨decompose-sequence#(op, st_0;out)⟩ true

iii-1 Right behaviour of *decompose-sequence#*, i.e. the state resulting from the execution of *decompose-sequence#* complies with the axiom that describes the function *decompose-sequence*:

$\vdash$

iscompop(sequence)

$\rightarrow$ (⟨decompose-sequence#(sequence, ini-state;out_0)⟩ out_0 = fin-state

$\rightarrow$ ($\forall$ id.id $\in_{ids}$ sequence.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id)))

The proofs of these obligations will ensure that *decompose-sequence#* is a correct implementation and that it satisfies the specification of *decompose-sequence*.

Both **i-1** and **iii-1** proofs have been carried out in KIV, by structural induction on operator. Structural induction makes direct use of the recursively described structure of data types. According to the basic form of structural induction, in order to prove that a property p is true for all defined terms we have to demonstrate that [Ehrig and Mahr, 1985]:

(1) the property p is true for all constant symbols.

(2) for each term $N(t_1, t_2, \ldots t_n)$ that can be constructed from the terms $t_1, t_2, \ldots t_n$, if $p(t_1), p(t_2), \ldots p(t_n)$ are true then also $p(N(t_1, t_2, \ldots t_n))$ is true.

In the proofs of **i-1** and **iii-1** we have used two lemmas establishing analogous obligations on *control#* procedure, i.e. the termination and the right behaviour of *control#*:

i-1-control Termination of *control#*:

$\vdash \langle \text{control\#(dec, st}_0\text{;out)} \rangle \, \text{true}$

iii-1-control Right behaviour of *control#*:

$$
\begin{array}{l}
\vdash \\
\quad \text{isprimop(op)} \\
\quad \vee \; (\quad \langle \text{control\#(op.dec, ini-state;out)} \rangle \, \text{out} = \text{fin-state} \\
\qquad \rightarrow (\forall \; \text{id.id} \in_{ids} \text{op.compinf.operoutput} \rightarrow \text{is-item-in-state(fin-state, id)}))
\end{array}
$$

The proofs of **i-1-control** and **iii-1-control** have also been carried out by structural induction. The proof of **iii-1-control** turned out to be particularly difficult. The proof tree in figure 7.12 serves to get an idea of the complexity of this proof. The details of the different steps can also be found in appendix D. The proof is based on several lemmas as the graph of lemma dependences in figure 7.13 depicts. Next we describe the most important ones.

The following lemma states that all the outputs of a compound operator must be included among the outputs of some suboperator in its decomposition. This lemma has been used twice in **iii-1-control** proof (in steps 4 and 43):

compop-output-is-in-dec Operator outputs are anywhere in its decomposition:

$$
\begin{array}{l}
\vdash \\
\quad \text{isprimop(op)} \\
\quad \vee \; (\forall \; \text{id.} \quad \text{id} \in_{ids} \text{op.compinf.operoutput} \\
\qquad\qquad \rightarrow (\exists \; \text{op}_0. \quad \text{indec(op}_0\text{, op.dec)} \\
\qquad\qquad\qquad\qquad \wedge \; (\quad \text{iscompop(op}_0) \wedge \text{id} \in_{ids} \text{op}_0\text{.compinf.operoutput} \\
\qquad\qquad\qquad\qquad\quad \vee \; \text{isprimop(op}_0) \wedge \text{id} \in_{ids} \text{op}_0\text{.priminf.operoutput})))
\end{array}
$$

A sequential decomposition is refined by *control#* in terms of the refinement or execution of the operators within it. This is reduced at the end to operator execution, the competence of which ensures that all the outputs appear in the resulting state. Then, in order to prove that after the termination of *control#* all the outputs of the compound operator appear in the final state it must be assured that they are all included in the outputs of some operator within the decomposition.

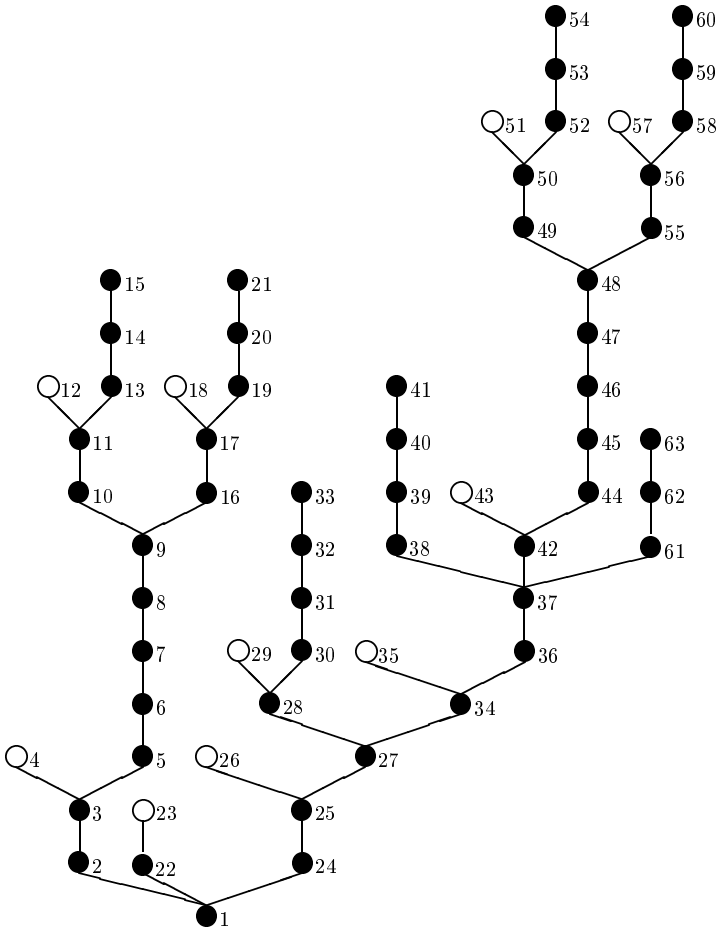


Figure 7.12: *Proof tree of* **iii-1-control**. Circles indicate proof steps; hollow circles mark the steps where an axiom or lemma has been inserted to close an open goal either directly (e.g. in step 23) or indirectly.

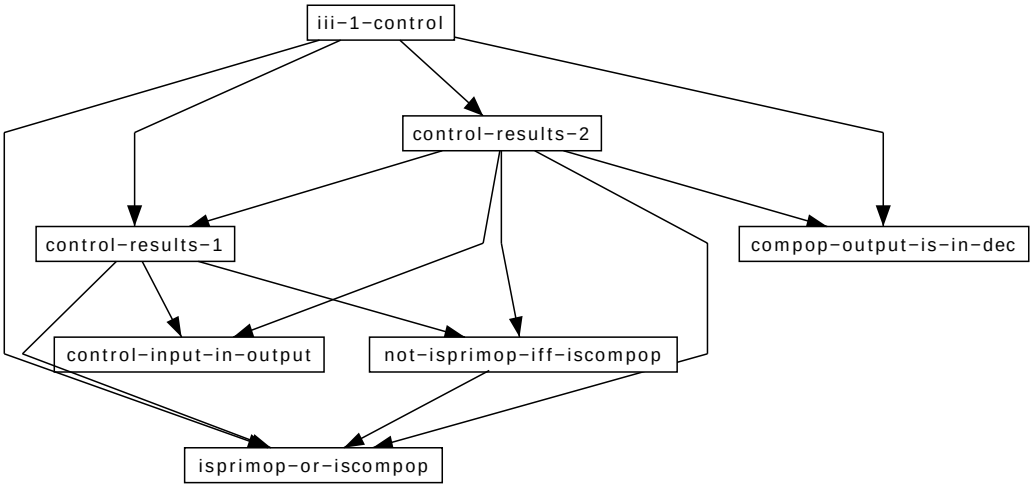


Figure 7.13: *Graph of lemma dependences*

In our simplified operator specification, data distribution is assumed to be established via argument/parameter identifiers. Therefore, the lemma **compop-output-is-in-dec** can be understood as a formalisation of a property related to the correctness of operator data distribution that we have identified in chapter 6. More specifically, it refers to the completeness of operator output distribution. This property cannot be proved from the specifications and therefore it must be ensured by other means. This is the kind of knowledge base property that an external verification procedure has to check.

The following lemmas state that after the termination of *control#* all the outputs of the suboperators in the decomposition are included in the final state. They have also been used in the proof of **iii-1-control** (e.g. in steps 12 and 18):

control-results-1 After the termination of *control#*, for all primitive operators in the decomposition it holds that all their outputs are included in the final state:

$\vdash$

$\langle$control#(dec, ini-state;out)$\rangle$ out = fin-state

$\rightarrow$ ($\forall$ op. indec(op, dec) $\wedge$ isprimop(op)

$\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.priminf.operoutput $\rightarrow$ is-item-in-state(fin-state, id)))

control-results-2 After the termination of *control#*, for all compound operators in the decomposition it holds that all their outputs are included in the final state:

$\vdash$

$\langle$control#(dec, ini-state;out)$\rangle$ out = fin-state

$\rightarrow$ ($\forall$ op. indec(op, dec) $\wedge$ iscompop(op)

$\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id)))

The proofs of **control-results-1** and **control-results-2** are based on the following lemma (see figure 7.13):

control-input-in-output After the termination of *control#*, all the elements in the initial state persist in the final state:

$\vdash$

$\langle$control#(dec, ini-state;out$_0$)$\rangle$ out$_0$ = fin-state

$\rightarrow$ ($\forall$ id.is-item-in-state(ini-state, id) $\rightarrow$ is-item-in-state(fin-state, id))

The refinement of a sequential decomposition is performed by the refinement or execution of its operators. For operator execution, a specification axiom ensures that all the operator

outputs appear in the resulting state. In order to prove that after refinement all the outputs of the operators in the decomposition are included in the final state, it must be ensured that operator refinement does not eliminate any item in the initial state.

A complete proof exists for **control-input-in-output**. In this proof we needed to ensure that *execute-operator* complies with an analogous property, for which purpose the corresponding axiom has been included in the specification *execute-operator-competence*. This persistence axiom concerns a property that the programs to be supervised should comply with. Although we are more interested in properties describing the knowledge base contents, this example illustrates the fact that the set of assumptions that a PS engine makes cannot completely be isolated, as we observed in chapter 5.

Finally, the following lemmas, which can easily be proved from the specification lemmas of *basic-operator*, have been included and repeatedly reused in proofs:

isprimop-or-iscompop An operator is either primitive or compound:

$\vdash$ isprimop(op) $\vee$ iscompop(op)

not-isprimop-iff-iscompop An operator is compound if and only if it is not primitive:

$\vdash \neg$ isprimop(op) $\leftrightarrow$ iscompop(op)

7.6 Conclusions

The application of software verification techniques is imperative to ensure the reliability of PS engines. We have presented a first experience in this direction which consists in the specification and verification of one of the subtasks identified in our knowledge model, implemented on the basis of other elementary instructions. This approach is well suited to the development process of PS engines within LAMA.

Our experiences in the specification and verification of compositions of instructions in the style of BLOCKS ones has demonstrated the utility of the application of software verification techniques for both the V&V of PS engines and the detection of the assumptions that they make. Indeed, the verification of the instruction in our experiments has required the introduction of a lemma expressing one of the properties that has been informally identified in chapter 6. Even limiting our experiments to simplified versions of data types and instructions, the time required to complete the specification, implementation and verification phases has been substantial[2]. In spite of this we believe that both the V&V of BLOCKS instructions

[2] Aproximately one month of work for a novice user, excluding training period.

and PS engines must be carried out in a principled way and not only by means of tests, as it is currently the case in LAMA. This formal verification has the additional benefit of supporting the process of detection of implementation assumptions.

The formal verification is a difficult job for which the support of an adequate tool is fundamental. Our assessment on KIV is very positive concerning the proof facilities that it integrates. In addition to this, the structuring of specifications and modularisation of software that KIV enforces, and their verification implications, are fundamental to tackle the V&V of PS engines. A negative aspect is that the KIV library of specifications lacks data types close to the structures that knowledge-based systems use to represent knowledge, e.g. rule bases.

Chapter 8

Conclusions

PROGRAM SUPERVISION SYSTEMS have proven to be a very useful support in the utilisation of program libraries by unexperienced users. In order to become reliable industrial applications, PS systems demand the application of V&V techniques. In this thesis we have presented our work on the verification and validation of PS systems, of both the PS knowledge bases and inference engines.

8.1 Contributions of the thesis

We have addressed different aspects of the verification and validation of PS systems by combining previous research results in several disciplines. Significant contributions of this thesis correspond to cross-fertilisations between the field of V&V of knowledge-based systems and other fields:

1. **Use of knowledge engineering techniques as basis for the V&V of knowledge bases.** In particular, the use of a knowledge-level analysis to understand knowledge utilisation by the inference engine, mainly the role that each piece of knowledge plays during reasoning. Exploiting this information to guide the knowledge acquisition process has proven to be a very powerful technique in the knowledge engineering field [David et al., 1993]. On the contrary, in the field of V&V of knowledge-based systems the emphasis has been traditionally put on symbol-level issues. We have shown that

exploiting the information on knowledge utilisation results in definitions of V&V issues which are much more meaningful since they use the terminology of the expert [Marcos et al., 1997].

2. **Use of existing knowledge base V&V techniques in the context defined above.** The field of V&V of knowledge-based systems provides us with a variety of symbol-level verification techniques. The V&V issues defined after a knowledge-level analysis have been reinterpreted into the corresponding symbol-level ones, and the appropriate techniques have been used in the implementation of our library of verification procedures. Thereby existing symbol-level techniques are used for the verification of knowledge-level issues.
3. **Use of a software verification tool for the V&V of inference engines and the identification of knowledge assumptions.** In the field of V&V of knowledge-based systems the inference engine is usually assumed to be correct. Recent work in the knowledge engineering area has demonstrated the utility of the V&V of problem-solving methods for the identification of the assumptions that they implicitly make on knowledge [Fensel and Schönegge, 1997]. In this line we have shown in our experiments with the KIV verification tool that the V&V of BLOCKS-based instructions can be used to formally identify the assumptions that they make on domain knowledge, which are the knowledge-level V&V issues that we emphasise in this thesis.
4. **Consider the V&V of knowledge-based systems in compositional development frameworks.** The V&V of PS knowledge bases and inference engines is viewed within the framework of the LAMA platform. Within LAMA inference engines can be modified (or even configured from scratch) to better suit the characteristics of an application domain. Although there exist many knowledge engineering frameworks in this line, this is not the usual hypothesis for the V&V of knowledge-based systems. Regarding the V&V of knowledge bases this has led us to analyse different methods for the PS subtasks, rather than a fixed monolithic PS method. According to the assumptions of the most frequent methods, a verification library has been implemented.

The previous contributions are partial aspects of our original approach to the V&V of PS systems. The main contributions of this thesis, related to the field of program supervision, are the **methodologies for the V&V of PS knowledge bases and inference engines**. Based on our model-based approach, the proposed methodology to the V&V of knowledge bases consists in:

- exploiting the knowledge model of the target PS inference engine to find out the prop-

erties that knowledge bases have to verify.

- generating the verification module adapted to these properties, from the verification library that we have implemented.

Additionally, we have sketched a methodology for the V&V of inference engines which consists in:

- specifying and verifying the BLOCKS components from which the PS inference engines are built.
- verifying that the PS inference engine configured by combining these components is correct, i.e. that it terminates and that it is adequate to solve the PS task.

The difficulty of the development of PS knowledge bases stems from the different types of knowledge used and from the multiple representations employed. Despite this, little effort has been dedicated to the development of V&V tools or methodologies adapted to the particularities of PS systems. An exception is the work for the syntactic verification of MVP knowledge bases in [Chien, 1996]. In this thesis we have shown that the V&V of PS systems can be tackled by the joint application of techniques in the fields of V&V of knowledge-based systems, knowledge engineering and software engineering. This approach is not constrained to the V&V of PS systems but it can be useful for other knowledge-based systems with similar characteristics.

8.1.1 Additional benefits

Our approach to the V&V of PS knowledge bases presupposes a phase of knowledge modeling. We have confirmed that the **knowledge modeling** that we have carried out has other interesting applications, besides serving as basis for the V&V of knowledge bases:

1. **to guide the knowledge acquisition of PS applications.** Our knowledge model provides us with a characterisation of PS engines in terms of the assumptions that they make on domain knowledge. This characterisation can be exploited to determine the adequacy of a PS engine to a target domain, which is very important when engineering new PS applications [Nunes de Barros et al., 1996].

2. **to guide the knowledge acquisition of PS inference engines.** Our knowledge model can guide the knowledge acquisition process necessary for the design of new PS inference engines. It has been exploited in this way for the design of MedIA engine, which was built by comparison with PEGASE and PULSAR, leading to reuse of parts of their methods and integration of specific features [Crubézy et al., 1998]. Thereby MedIA design resulted in an increased/refined knowledge model integrating new knowledge concepts and methods.
3. **to identify other usable and reusable** BLOCKS **components.** The BLOCKS library provides a set of small-grain sized components so that skilled designers are able to fine-tune the behaviour of an engine. On the other hand, big-grain components would allow for an efficient reuse for less-skilled designers [Crubézy et al., 1998]. Our knowledge model has helped identify typical methods that could be reused in new engines, and therefore constitute good candidates for other BLOCKS components.

In addition to the benefits specific to our modeling work, we acknowledge that in general a knowledge model can be used **to support the V&V of inference engines** by the expert since it constitutes a functional specification that uses terms closer to the expert's discourse than implementation [David et al., 1993].

8.2 Limitations of the results

8.2.1 Limitations of the model-based verification of knowledge bases

According to the methodology that we propose, the verification modules adapted to new PS engines are configured after a knowledge-level analysis, by using the verification library that we have implemented. Besides, these verification modules are built manually.

Two main drawbacks are inherent to this approach:

- Incompleteness of our verification library. Engines implying important modifications on the PS domain model and/or methods studied in this thesis have to be reconsidered, i.e. analysed in order to implement the necessary verification procedures. It can be argued that assuming that our verification library can be complete is unrealistic. However, we believe that given the specialised task we focus on, after various passages by knowledge modeling-refinement of the model steps, our verification library will include most of the necessary verification procedures.

- Manual work. Both the knowledge-level analysis of PS engines, with the subsequent identification of the model-based V&V issues, and the configuration of the appropriate verification modules imply considerable manual work. The former, which is fundamental to our approach, can be partially supported by the utilisation of software verification techniques.

8.2.2 Limitations of the verification and validation of inference engines

We propose using a traditional software verification tool for the V&V of inference engines. The KIV verification tool, which we have used in our experiments, can be used for the specification and verification of the components in the BLOCKS library. KIV can be used as well for the V&V of new engines, by proving the correctness of algorithmic compositions of BLOCKS components.

The disadvantages of the proposed approach are:

- The difficulty of the specification and verification of realistic versions of BLOCKS components. Indeed our experiences have been limited to e.g. simplified data types for this reason.
- The interactive nature of proofs in KIV. This can be seen as a disadvantage, but we believe that it is unavoidable, especially if we intend to support the process of assumption detection with software V&V.

Formal software verification is a difficult task, but it is imperative to ensure the reliability of PS engines. It has the additional benefit of supporting the detection process of the assumptions that the PS engine makes on domain knowledge, which are fundamental in our approach to the V&V of PS knowledge bases.

Finally, it is important to mention the problem of the operationalisation of verified implementations. In order to ensure the reliability of verified (compositions of) components, their operationalisation in LAMA should preserve the structure of their implementation in KIV.

8.3 Perspectives

The work in this thesis has revealed other interesting work perspectives. The information on knowledge utilisation during reasoning dictates exactly what knowledge is necessary. The

model-based V&V issues that we have exploited in this thesis consider knowledge utilisation and therefore constitute the precise properties that the knowledge base should verify rather than the traditionally used symbol-level ones. The same principle is applicable to other important development activities, e.g. knowledge acquisition and knowledge refinement. In addition, significantly different work is necessary in order to improve the configuration of model-based V&V tools. According to this we have identified the following work directions:

1. **Model-based knowledge acquisition.** The information on knowledge utilisation can be directly applied to guide the knowledge acquisition process. Many systems/frameworks have demonstrated that model-based knowledge acquisition results in tools much easier to use than tools at the symbol-level. Role-limiting methods [Marcus, 1988] and PROTÉGÉ [Musen, 1989] are (single-method) knowledge-based systems with dedicated knowledge acquisition tools. PROTÉGÉ-II [Puerta et al., 1992] and DIDS [Runkel and Birmingham, 1995] are multiple-method frameworks for the generation of knowledge acquisition tools. Par-KAP [Nunes de Barros et al., 1997] is a framework that supports knowledge acquisition in planning in a different manner: it helps find the knowledge requirements of a particular method or the applicable methods given a domain specification.
2. **Model-based knowledge refinement.** When a knowledge base prevents the system from finding a (correct) solution to a problem that the expert estimates solvable, the faulty piece of knowledge must be identified. In this case, tracking the problem-solving tasks that are performed appears to be much more efficient than using debugging facilities at the symbol level [David et al., 1993]. An example is the completion analysis tool of MVP [Chien, 1996] implicitly makes use of the employed planning representations and tasks to facilitate the process of isolating bugs.
3. **Automatic configuration of model-based V&V/knowledge acquisition tools.** The generation of model-based tools adapted to a particular method can be automated. This usually implies the explicit representation of domain models and methods, and of their corresponding verification/knowledge acquisition implications. The PROTÉGÉ-II and DIDS frameworks automatically generate knowledge acquisition tools from methods configured using small-grain sized components. Different models exist for the automatic generation of knowledge acquisition tools [Eriksson and Musen, 1993]. For instance, a set of strategies that provide a global view of knowledge acquisition is used in DIDS [Runkel and Birmingham, 1993].

Appendix A

Definitions of Rule-Based Anomalies

Traditional anomalies for knowledge-based systems concern properties such as redundant, contradictory or deficient knowledge. The formulation of the most common anomalies for rule bases that follows has been extracted from [Preece and Shinghal, 1994] and [van Harmelen and Aben, 1995]. We next introduce the terminology and notation used in the formulations:

- A rule R_i is formula of the form $l_1 \wedge \cdots \wedge l_n \rightarrow m$ where each l_i and m are first order literals.
- For each rule $R_i = l_1 \wedge \cdots \wedge l_n \rightarrow m$, we write $antec(R) = l_1 \wedge \cdots \wedge l_n$ and $conseq(R) = m$.
- A rule set $\mathcal{R}$ is a set of rules.
- The goal-literals $\mathcal{G}$ is the set of all ground literals that could possibly be output from the rule set.
- The input-literals $\mathcal{I}$ is the set of all ground literals that constitute all possible inputs to the rule set.
- A semantic constraint is a set of literals $\{l_1, \ldots, l_n\}$ such that their conjunction $l_1 \wedge \cdots \wedge l_n$ is regarded as a semantic inconsistency (e.g. the set $\{male(x),\ pregnant(x)\}$). We write $\mathcal{C}$ for the set of all semantic constraints for a rule set.
- An enviroment is a subset of $\mathcal{I}$ that does not imply any semantic constraint. We write $\mathcal{E}$ for the set of all such environments. Formally, $e \nvdash c$ for all $e \in \mathcal{E}$ and all $c \in \mathcal{C}$.

The formulation of rule-based anomalies follows:

Unsatisfiable rule. A rule R is *unsatisfiable iff* there is no way of deducing R's antecedent from any legal input:

$$\neg(\exists e \in \mathcal{E}, \exists \sigma : R \cup e \vdash \sigma \circ antec(R)).$$

(we write $\sigma \circ \phi$ for the application of a substitution σ to a formula ϕ).

Unusable rule. A rule R is *unusable iff* the consequent of R subsumes neither a goal literal noy any antecedent literal in the rule set:

$$\begin{aligned}\forall \sigma : (&\sigma \circ conseq(R) \notin \mathcal{G} \wedge \\ &\neg \exists R' \in \mathcal{R} \setminus \{R\} : \sigma \circ conseq(R) \in antec(R')).\end{aligned}$$

Subsumed rule. A rule R is *subsumed iff* there exists a more general rule:

$$\exists R' \in \mathcal{R} \setminus \{R\}, \exists \sigma : R' \rightarrow \sigma \circ R.$$

Redundant rule. A rule R is *redundant* in a rule set $\mathcal{R}$ *iff* R is not essential for the computation of any literal from any environment:

$$\forall e \in \mathcal{E}, \forall h : \text{if } \mathcal{R} \cup e \vdash h \text{ then } \mathcal{R} \setminus \{R\} \cup e \vdash h.$$

Inconsistent rule pair. Rules R and R' are an *inconsistent pair iff* R and R' are both applicable and derive a semantic constraint:

$$\begin{aligned}\exists e \in \mathcal{E}, \exists \sigma, \exists \sigma' : &\mathcal{R} \cup e \vdash \sigma \circ conseq(R) \wedge \\ &\mathcal{R} \cup e \vdash \sigma' \circ conseq(R') \wedge \\ &\{\sigma \circ conseq(R),\ \sigma' \circ conseq(R')\} \in \mathcal{C}.\end{aligned}$$

Inconsistent rule set. A rule set $\mathcal{R}$ is *inconsistent iff* from some legal input it is possible to derive a semantic constraint from $\mathcal{R}$:

$$\exists e \in \mathcal{E}, \exists c \in \mathcal{C} \ : \ \mathcal{R} \cup e \vdash c.$$

Circular rule set. A rule set $\mathcal{R}$ is *circular iff* $antec(R)$ cannot be derived from any environment, except by adding R's consequent:

$$\exists R \in \mathcal{R} \ : \forall e \in \mathcal{E} : \mathcal{R} \cup e \nvdash antec(R) \wedge$$
$$\exists e \in \mathcal{E} : \mathcal{R} \cup e \cup conseq(R) \vdash antec(R).$$

Unused input. An input literal $i \in \mathcal{I}$ is *unused iff* any result that can be computed from any environment can also be computed from that environment minus i:

$$\forall e \in \mathcal{E} \ \forall g \in \mathcal{G} \ : \text{ if } \mathcal{R} \cup e \vdash g \text{ then } \mathcal{R} \cup e \setminus \{i\} \vdash g.$$

Incomplete rule set. A rule set $\mathcal{R}$ is *incomplete iff* there exists some output that cannot be computed from any environment:

$$\exists e \in \mathcal{E} : \forall g \in \mathcal{G} : \mathcal{R} \cup e \nvdash g.$$

Appendix B

Task Knowledge of PEGASE and PULSAR

Here we present a complete description of PEGASE and PULSAR task knowledge. In this description we use the CML (the CommonKADS Conceptual Modelling Language) textual notation [Wielinga et al., 1994]. The CML description of a task has two parts: a task definition and a task body. The task definition has the following subparts:

- goal, or textual description of the goal that can be achieved through the application of the task.
- input and output, which is a definition of the roles that the task manipulates.

The task body consists of:

- task type (composite or primitive).
- decomposition, or subtasks that the task decomposes into.
- additional roles, which are additional data stores that are introduced by the decomposition.
- control structure, or description of the control over the subtasks to achieve the task. A usual way of describing the control structure is procedural pseudo-code.

PEGASE

task PEG supervise;
task-definition
goal: "Perform program supervision";
input: problem specification;
operator KB;
output: result evaluation;
final plan;
final state;
task-body
type: composite;
sub-tasks: initialise state, initialise plan, initialise functionality, PEG plan and execute;
additional-roles: initial state;
initial state history;
initial plan;
initial functionality;
control-structure:

initialise state (problem specification→ initial state+initial state history)
initialise plan (problem specification→ initial plan)
initialise functionality (problem specification→ initial functionality)
PEG plan and execute (initial plan+initial state+initial state history+initial functionality+operator KB→ result evaluation+final plan+final state)

task PEG plan and execute;
task-definition
goal: "Perform operator-based planning";
input: plan;
state;
state history;
functionality;
operator KB;
output: result evaluation;
executed plan;
final state;
task-body
type: composite;
sub-tasks: PEG expand plan, select plan, PEG execute plan;
additional-roles: expanded plans;
expanded plan;
control-structure:

PEG expand plan (plan+state+functionality+operator KB→ expanded plans)
while result evaluation <> success **and** expanded plans <> ∅ **do**
select plan (expanded plans→ expanded plan+expanded plans)
PEG execute plan (expanded plan+state+state history→ result evaluation+executed plan+final state+state history)
end while
if expanded plans = ∅ **then**
result evaluation← all-plan-expansions-failed
end if

task PEG expand plan;
task-definition
goal: "Expand a plan";
input: plan;
state;
functionality;
operator KB;
output: expanded plans;
task-body
type: composite;
sub-tasks: PEG select and order operators, PEG integrate operators;
additional-roles: ordered selected operators;
control-structure:
PEG select and order operators (state+functionality+operator KB→ ordered selected operators)
PEG integrate operators (plan+ordered selected operators→ expanded plans)

task PEG select and order operators;
task-definition
goal: "Select operators from the operator KB and order them";
input: state;
functionality;
operator KB;
output: ordered selected operators;
task-body
type: composite;
sub-tasks: PEG select operators, PEG order operators;
additional-roles: selected operators;
control-structure:
PEG select operators (state+functionality+operator KB→ selected operators)
PEG order operators (state+selected operators→ ordered selected operators)

task PEG execute plan;
task-definition
goal: "Execute operators in a plan";
input: plan;
state;
state history;
output: result evaluation;
executed plan;
final state;
state history;
task-body
type: composite;
sub-tasks: select first plan operator, PEG refine, execute and repair operator;
additional-roles: unified operator;
control-structure:
select first plan operator (plan+state→ unified operator)
PEG refine, execute and repair operator (unified operator+plan+state+state history→ result evaluation+executed plan+final state+state history)

task	select first plan operator;
task-definition	
goal:	"Select an operator from a plan";
input:	plan;
	state;
output:	unified operator;
task-body	
type:	composite;
sub-tasks:	get plan operators, next, unify arguments;
additional-roles:	operators;
control-structure:	

get plan operators (plan→ operators)
next (operators→ unified operator)
unify arguments (unified operator+state→ unified operator)

task PEG refine, execute and repair operator;
task-definition
goal: "Perform hierarchical planning";
input: operator;
plan;
state;
state history;
output: result evaluation;
final plan;
final state;
state history;
task-body
type: composite;
sub-tasks: rule-based initialise parameters, PEG execute operator, update plan, PEG specialise operator, PEG decompose sequence, apply effects, rule-based evaluate results, PEG rule-based repair operator;
additional-roles: initialised operator;
evaluated operator;
control-structure:

rule-based initialise parameters (operator+state→ initialised operator)
if primitive operator **then**
PEG execute operator (initialised operator+state+state history→ final state+state history)
update plan (plan+initialised operator→ final plan)
else if specialisation **then**
PEG specialise operator (initialised operator+plan+state+state history→ final plan+final state+state history)
else if sequence **then**
PEG decompose sequence (initialised operator+plan+state+state history→ final plan+final state+state history)
end if
apply effects (initialised operator+final state→ final state)
rule-based evaluate results (operator+final state→ evaluated operator+result evaluation)
while result evaluation = repair **do**
PEG rule-based repair operator (evaluated operator+final state+state history→ final state+state history)
rule-based evaluate results (operator+final state→ evaluated operator+result evaluation)
end while

task rule-based initialise parameters;
task-definition
goal: "Initialise operator parameters";
input: operator;
state;
output: initialised operator;
task-body
type: composite;
sub-tasks: get oper initialisation RB, get RB next rule, test rule preconditions,
initialise par with action;
additional-roles: initialisation RB;
initialisation rule;
result;
control-structure:

get oper initialisation RB (operator→ initialisation RB)
get RB next rule (initialisation RB→ initialisation rule)
initialised operator← operator
while initialisation rule <> ∅ **do**
test rule preconditions (initialisation rule+state→ result)
if result = success **then**
initialise par with action (initialised operator+initialisation rule+state→ initialised operator)
end if
get RB next rule (initialisation RB→ initialisation rule)
end while

task PEG execute operator;
task-definition
goal: "Execute primitive operator";
input: operator;
state;
state history;
output: final state;
state history;
task-body
type: composite;
sub-tasks: prepare arguments, test oper preconditions, make program call,
call program, test oper postconditions, update hist with exec;
additional-roles: prepared operator;
result;
program call;
control-structure:

prepare arguments (operator→ prepared operator)
test oper preconditions (prepared operator+state→ result)
if result <> success **then**
warn ("invalid preconditions")
end if
make program call (prepared operator→ program call)
call program (program call)
test oper postconditions (prepared operator+state→ result)
if result <> success **then**
warn ("invalid postconditions")
end if
update hist with exec (prepared operator+final state+state history→ state history)

task PEG specialise operator;
task-definition
goal: "Perform hierarchical refinement of a specialisation";
input: operator;
plan;
state;
state history;
output: final plan;
final state;
state history;
task-body
type: composite;
sub-tasks: rule-based choose suboperators, update hist with spec, sort suboperators, next, distribute in arguments, PEG refine, execute and repair operator, distribute out arguments, update hist with exec;
additional-roles: suboperators;
suboperator;
result evaluation;
intermediate state;
control-structure:

rule-based choose suboperators (operator+state→ suboperators)
update hist with spec (operator+state history→ state history)
sort suboperators (suboperators→ suboperators)
next (suboperators→ suboperator)
distribute in arguments (operator+suboperator→ suboperator)
PEG refine, execute and repair operator (suboperator+plan+state+state history→ result evaluation+final plan+intermediate state+state history)
distribute out arguments (operator+intermediate state+final state→ final state)
update hist with exec (final state+state history→ state history)

task rule-based choose suboperators;
task-definition
goal: "Choose suboperators";
input: operator;
state;
output: suboperators;
task-body
type: composite;
sub-tasks: get oper choice RB, get RB next rule, test rule preconditions, choose suboper with action;
additional-roles: choice RB;
choice rule;
result;
control-structure:

```
get oper choice RB (operator→ choice RB)
get RB next rule (choice RB→ choice rule)
suboperators← ∅
while choice rule <> ∅ do
  test rule preconditions (choice rule+state→ result)
  if result = success then
    choose suboper with action (choice rule+suboperators→ suboperators)
  end if
  get RB next rule (choice RB→ choice rule)
end while
```

task PEG decompose sequence;
task-definition
goal: "Perform hierarchical refinement of a sequence";
input: operator;
plan;
state;
state history;
output: final plan;
final state;
state history;
task-body
type: composite;
sub-tasks: get oper suboperators, update hist with dec, next, bind in arguments, test subop applicability, PEG refine, execute and repair operator, distribute out arguments, update hist with exec;
additional-roles: suboperators;
suboperator;
result evaluation;
intermediate state;
control-structure:

get oper suboperators (operator→ suboperators)
update hist with dec (operator+state history→ state history)
intermediate state← ∅
repeat
next (suboperators→ suboperator)
bind in arguments (operator+suboperator+intermediate state→ suboperator)
if optional **then**
test subop applicability (suboperator+state→ result)
end if
if not optional **or** result = success **then**
PEG refine, execute and repair operator (suboperator+plan+state+state history→ result evaluation+final plan+intermediate state+state history)
distribute out arguments (operator+intermediate state+final state→ final state)
state← final state
plan← final plan
end if
until result evaluation <> continue **or** suboperators = ∅
update hist with exec (final state+state history→ state history)

task rule-based evaluate results;
task-definition
goal: "Evaluate results";
input: operator;
state;
output: evaluated operator;
result evaluation;
task-body
type: composite;
sub-tasks: get oper evaluation RB, get RB next rule, test rule preconditions, evaluate oper with action;
additional-roles: evaluation RB;
evaluation rule;
result;
control-structure:

get oper evaluation RB (operator→ evaluation RB)
get RB next rule (evaluation RB→ evaluation rule)
evaluated operator← operator
while evaluation rule <> ∅ **do**
test rule preconditions (evaluation rule+state→ result)
if result = success **then**
evaluate oper with action (evaluated operator+evaluation rule→ evaluated operator)
end if
get RB next rule (evaluation RB→ evaluation rule)
end while
get oper result evaluation (evaluated operator→ result evaluation)

task PEG rule-based repair operator;
task-definition
goal: "Repair operator";
input: operator;
state;
state history;
output: final state;
state history;
task-body
type: composite;
sub-tasks: get oper repair RB, get RB next rule, test rule preconditions,
PEG repair oper with action;
additional-roles: repair RB;
repair rule;
result;
control-structure:

get oper repair RB (operator→ repair RB)
get RB next rule (repair RB→ repair rule)
result← failure
while result <> success **and** repair rule <> ∅ **do**
test rule preconditions (repair rule+state→ result)
if result = success **then**
PEG repair oper with action (operator+repair rule+state+state history→ final state+state history)
end if
get RB next rule (repair RB→ repair rule)
end while

task PEG repair operator with action;
task-definition
goal: "Repair operator with repair action";
input: operator;
repair rule;
state;
state history;
output: final state;
state history;
task-body
type: composite;
sub-tasks: get rule action, rule-based adjust parameters, PEG execute operator,
repair with send, repair with sendup, repair with backchoice;
additional-roles: action;
action argument;
error arguments;
adjusted operator;
control-structure:

get rule action (repair rule→ action+action argument+error arguments)
if action = re-execute **then**
rule-based adjust parameters (operator+state→ adjusted operator)
PEG execute operator (adjusted operator+state+state history→ final state+state history)
else if action = send-down **or** action = send-op **then**
repair with send (action argument+error arguments+state history→ final state+state history)
else if action = send-up **then**
repair with sendup (operator+error arguments+state history→ final state+state history)
else if action = back-choice **then**
repair with backchoice (operator+error arguments+state history→ final state+state history)
end if

task rule-based adjust parameters;
task-definition
goal: "Adjust operator parameters";
input: operator;
state;
output: adjusted operator;
task-body
type: composite;
sub-tasks: get oper adjustment RB, get RB next rule, test rule preconditions, adjust par with action;
additional-roles: adjust RB;
adjustment rule;
result;
control-structure:

get oper adjustment RB (operator→ adjustment RB)
get RB next rule (adjustment RB→ adjustment rule)
adjusted operator← operator
while adjustment rule <> ∅ **do**
test rule preconditions (adjustment rule+state→ result)
if result = success **then**
adjust par with action (adjusted operator+adjustment rule→ adjusted operator)
end if
get RB next rule (adjustment RB→ adjustment rule)
end while

task repair with send;
task-definition
goal: "Repair operator with send";
input: action argument;
error arguments;
state history;
output: final state;
state history;
task-body
type: composite;
sub-tasks: change context, modify oper assessment, PEG rule-based repair operator;
additional-roles: target operator;
target state;
control-structure:

change context (action argument+state history→ target operator+target state+state history)
modify oper assessment (target operator+error arguments→ target operator)
PEG rule-based repair operator (target operator+target state+state history→ final state+state history)

task repair with sendup;
task-definition
goal: "Repair operator with sendup";
input: operator;
error arguments;
state history;
output: final state;
state history;
task-body
type: composite;
sub-tasks: get hist father, change context, modify oper assessment,
PEG rule-based repair operator;
additional-roles: action argument;
target operator;
target state;
control-structure:

get hist father (operator+state history→ action argument)
change context (action argument+state history→ target operator+target state+state history)
modify oper assessment (target operator+error arguments→ target operator)
PEG rule-based repair operator (target operator+target state+state history→ final state+state history)

task repair with backchoice;
task-definition
goal: "Repair operator with backchoice;
input: operator;
error arguments;
state history;
output: final state;
state history;
task-body
type: composite;
sub-tasks: get hist last choice, change context, modify oper assessment,
PEG rule-based repair operator;
additional-roles: action argument;
target operator;
target state;
control-structure:

get hist last choice (operator+state history→ action argument)
change context (action argument+state history→ target operator+target state+state history)
modify oper assessment (target operator+error arguments→ target operator)
PEG rule-based repair operator (target operator+target state+state history→ final state+state history)

PULSAR

task PUL supervise;
task-definition
goal: "Perform program supervision";
input: problem specification;
operator KB;
output: result evaluation;
final plan;
final state;
task-body
type: composite;
sub-tasks: initialise state, initialise plan, initialise constraints, PUL plan and execute;
additional-roles: initial state history;
initial state;
initial plan;
initial constraints;
control-structure:

initialise state (problem specification→ initial state+initial state history)
initialise plan (problem specification→ initial plan)
initialise constraints (problem specification→ initial constraints)
PUL plan and execute (initial plan+initial state+initial state history+initial constraints+operator KB→ result evaluation+final plan+final state)

task PUL plan and execute;
task-definition
goal: "Perform operator-based planning";
input: plan;
state;
state history;
constraints;
operator KB;
output: result evaluation;
executed plan;
final state;
task-body
type: composite;
sub-tasks: PUL expand plan, select plan, PUL execute plan, PUL plan and execute;
additional-roles: expanded plans;
expanded plan;
control-structure:

if constraints ∈ state **then**
executed plan← plan
final state← state
result evaluation← success
else
PUL expand plan (plan+state+constraints+operator KB→ expanded plans)
while result evaluation <> success **and** expanded plans <> ∅ **do**
select plan (expanded plans→ expanded plan+expanded plans)
PUL execute plan (expanded plan+state+state history→ result evaluation+executed plan+final state+state history)
if result evaluation = success **then**
PUL plan and execute (executed plan+final state+state history+constraints+operator KB→ result evaluation+executed plan+final state)
end if
end while
if expanded plans = ∅ **then**
result evaluation← all-plan-expansions-failed
end if
end if

task PUL expand plan;
task-definition
goal: "Expand a plan";
input: plan;
state;
constraints;
operator KB;
output: expanded plans;
task-body
type: composite;
sub-tasks: PUL select and order operators, PUL integrate operators;
additional-roles: ordered selected operators;
control-structure:

PUL select and order operators (plan+state+constraints+operator KB→ ordered selected operators)
PUL integrate operators (plan+ordered selected operators→ expanded plans)

task PUL select and order operators;
task-definition
goal: "Select operators from the operator KB and order them";
input: plan;
state;
constraints;
operator KB;
output: ordered selected operators;
task-body
type: composite;
sub-tasks: PUL select operators, PUL order operators;
additional-roles: selected operators;
control-structure:

PUL select operators (plan+state+constraints+operator KB→ selected operators)
PUL order operators (plan+state+selected operators→ ordered selected operators)

task PUL execute plan;
task-definition
goal: "Execute operators in a plan";
input: plan;
state;
state history;
output: result evaluation;
executed plan;
final state;
state history;
task-body
type: composite;
sub-tasks: select applicable plan operator, PUL refine, execute and repair operator;
additional-roles: unified operator;
control-structure:

result evaluation← success
select applicable plan operator (plan+state→ unified operator)
while unified operator <> ∅ **and** result evaluation = success **do**
PUL refine, execute and repair operator (unified operator+plan+state+state history→ result evaluation+executed plan+final state+state history)
plan← executed plan
state← final state
select applicable plan operator (plan+state→ unified operator)
end while

task select applicable plan operator;
task-definition
goal: "Select an operator from a plan";
input: plan;
state;
output: unified operator;
task-body
type: composite;
sub-tasks: get plan operators, next, test oper preconditions, unify arguments;
additional-roles: operators;
result;
control-structure:

get plan operators (plan→ operators)
next (operators→ unified operator)
test oper preconditions (unified operator+state→ result)
while result <> success **do**
next (operators→ unified operator)
test oper preconditions (unified operator+state→ result)
end while
unify arguments (unified operator+state→ unified operator)

task PUL refine, execute and repair operator;
task-definition
goal: "Perform hierarchical planning";
input: operator;
plan;
state;
state history;
output: result evaluation;
final plan;
final state;
state history;
task-body
type: composite;
sub-tasks: rule-based initialise parameters, PUL execute operator, update plan,
PUL specialise operator, PUL decompose sequence,
rule-based evaluate results, PUL rule-based repair operator;
additional-roles: initialised operator;
evaluated operator;
control-structure:

rule-based initialise parameters (operator+state→ initialised operator)
if primitive operator **then**
PUL execute operator (initialised operator+state+state history→ final state+state history)
update plan (plan+initialised operator→ final plan)
else if specialisation **then**
PUL specialise operator (initialised operator+plan+state+state history→ final plan+final state+state history)
else if sequence **then**
PUL decompose sequence (initialised operator+plan+state+state history→ final plan+final state+state history)
end if
rule-based evaluate results (operator+final state→ evaluated operator+result evaluation)
while result evaluation = repair **do**
PUL rule-based repair operator (evaluated operator+final state+state history→ final state+state history)
rule-based evaluate results (operator+final state→ evaluated operator+result evaluation)
end while

task PUL execute operator;
task-definition
goal: "Execute primitive operator";
input: operator;
state;
state history;
output: final state;
state history;
task-body
type: composite;
sub-tasks: prepare arguments, make program call,
call program, apply effects, apply data dependences, update hist with exec;
additional-roles: prepared operator;
program call;
control-structure:

prepare arguments (operator→ prepared operator)
make program call (prepared operator→ program call)
call program (program call)
apply effects (prepared operator+final state→ final state)
apply data dependences (final state→ final state)
update hist with exec (prepared operator+final state+state history→ state history)

task PUL specialise operator;
task-definition
goal: "Perform hierarchical refinement of a specialisation";
input: operator;
plan;
state;
state history;
output: final plan;
final state;
state history;
task-body
type: composite;
sub-tasks: rule-based choose suboperators, update hist with spec, next,
distribute in arguments, PUL refine, execute and repair operator,
distribute out arguments, update hist with exec;
additional-roles: suboperators;
suboperator;
result;
result evaluation;
intermediate state;
control-structure:

rule-based choose suboperators (operator+state→ suboperators)
update hist with spec (operator+state history→ state history)
intermediate state← ∅
repeat
next (suboperators→ suboperator)
distribute in arguments (operator+suboperator→ suboperator)
test oper preconditions (suboperator+state→ result)
if result = success **then**
PUL refine, execute and repair operator (suboperator+plan+state+state history→ result evaluation+final plan+intermediate state+state history)
if result evaluation = continue **then**
distribute out arguments (operator+intermediate state+final state→ final state)
end if
end if
until result evaluation = continue **or** suboperators = ∅
update hist with exec (final state+state history→ state history)

task PUL decompose sequence;
task-definition
goal: "Perform hierarchical refinement of a sequence";
input: operator;
plan;
state;
state history;
output: final plan;
final state;
state history;
task-body
type: composite;
sub-tasks: get oper suboperators, update hist with dec, next, bind in arguments, test oper preconditions, PUL refine, execute and repair operator, distribute out arguments, rule-based evaluate results, PUL rule-based repair operator, update hist with exec;
additional-roles: suboperators;
intermediate state;
suboperator;
result;
intermediate result evaluation;
result evaluation;
control-structure:

get oper suboperators (operator→ suboperators)
update hist with dec (operator+state history→ state history)
intermediate state← ∅
repeat
next (suboperators→ suboperator)
bind in arguments (operator+suboperator+intermediate state→ suboperator)
test oper preconditions (suboperator+state→ result)
if result = success **then**
PUL refine, execute and repair operator (suboperator+plan+state+state history→ result evaluation+final plan+intermediate state+state history)
distribute out arguments (operator+intermediate state+final state→ final state)
rule-based evaluate results (operator+final state→ evaluated operator+intermediate result evaluation)
while intermediate result evaluation = repair **do**
PUL rule-based repair operator (operator+final state+state history→ final state+state history)
rule-based evaluate results (operator+intermediate state→ evaluated operator+intermediate result evaluation)
end while
state← final state
plan← final plan
end if
until result <> success **or** intermediate result evaluation <> continue **or** suboperators = ∅
update hist with exec (final state+state history→ state history)

task PUL rule-based repair operator;
task-definition
goal: "Repair operator";
input: operator;
state;
state history;
output: final state;
state history;
task-body
type: composite;
sub-tasks: get oper repair RB, get RB next rule, test rule preconditions,
PUL repair oper with action;
additional-roles: repair RB;
repair rule;
result;
control-structure:

get oper repair RB (operator→ repair RB)
get RB next rule (repair RB→ repair rule)
result← failure
while result <> success **and** repair rule <> ∅ **do**
test rule preconditions (repair rule+state→ result)
if result = success **then**
PUL repair oper with action (operator+repair rule+state+state history→ final state+state history)
end if
get RB next rule (repair RB→ repair rule)
end while

task PUL repair operator with action;
task-definition
goal: "Repair operator with repair action";
input: operator;
repair rule;
state;
state history;
output: final state;
state history;
task-body
type: composite;
sub-tasks: get rule action, rule-based adjust parameters, PUL execute operator, repair with senddown;
additional-roles: action;
action argument;
error arguments;
adjusted operator;
control-structure:

get rule action (repair rule→ action+action argument+error arguments)
if action = re-execute **then**
rule-based adjust parameters (operator+state→ adjusted operator)
PUL execute operator (adjusted operator+state+state history→ final state+state history)
else if action = send-down **then**
repair with senddown (action argument+error arguments+state history→ final state+state history)
end if

task repair with senddown;
task-definition
goal: "Repair operator with send-down";
input: action argument;
error arguments;
state history;
output: final state;
state history;
task-body
type: composite;
sub-tasks: change context, modify oper assessment, PEG rule-based repair operator;
additional-roles: target operator;
target state;
control-structure:

change context (action argument+state history→ target operator+target state+state history)
modify oper assessment (target operator+error arguments→ target operator)
PUL rule-based repair operator (target operator+target state+state history→ final state+state history)

Appendix C

Algorithms for the Verification of Program Supervision Knowledge Bases

In the following we present the algorithms of the procedures for checking PS knowledge bases for important model-based properties. In the description of the algorithms we use a procedural notation including the most common programming constructs (if-then-else construct, for loop, etc). Except when we employ the italic font, the algorithms corresponding to the different calls are always described next to the procedure where they appear. The calls in italic font usually make reference to simple procedures which do not need any further explanation.

Data distribution correctness in specialisation decompositions

```
verify-specialisation-dist-correctness(op→ result):
begin
all-op-inputs-distributed(op→ aux-result1)
if not aux-result1 then
  result ← false
else
  all-op-outputs-receive-dist-1-value-per-subop(op→ aux-result2)
  if not aux-result2 then
    result ← false
  else
    all-subops-inputs-receive-dist-value(op→ aux-result3)
    if not aux-result3 then
      result ← false
    else
      result ← true
    end if
  end if
end if
end

all-op-inputs-distributed(op→ result):
op-input-arguments(op→ in-args)
for all in-arg such that in-arg∈in-args do
  distributes-in(in-arg→ aux-result)
  if not aux-result then
    error("No distribution of input of operator")
    result ← false
    exit
  end if
end for
result ← true
end

all-op-outputs-receive-dist-1-value-per-subop(op→ result):
begin
op-output-arguments(op→ out-args)
for all out-arg such that out-arg∈out-args do
  op-output-receives-dist(op+out-arg→ aux-result)
  if not aux-result then
    result ← false
    exit
  end if
end for
result ← true
end
```

```
op-output-receives-dist(op+out-arg→ result):
begin
subop-list(op→ subops)
for all subop such that subop∈subops do
  receives-out-dist-from(out-arg+subop→ aux-result)
  if not aux-result then
    error("No distribution for output from suboperator")
    result ← false
    exit
  end if
end for
result ← true
end

all-subops-inputs-receive-dist-value(op→ result):
begin
subop-list(op→ subops)
for all subop such that subop∈subops do
  all-subop-inputs-receive-dist(subop→ aux-result)
  if not aux-result then
    result ← false
    exit
  end if
end for
result ← true
end

all-subop-inputs-receive-dist(subop→ result):
begin
op-input-arguments(subop→ in-args)
for all in-arg such that in-arg∈in-args do
  receives-in-dist(in-arg→ aux-result)
  if not aux-result then
    error("No distribution for input of suboperator")
    result ← false
    exit
  end if
end for
result ← true
end
```

Data distribution correctness in sequential decompositions (with optional operators)

```
verify-dist-correctness(op→ result):
begin
has-optional-subops(op→ aux-result1)
if not aux-result1 then {there are not optional suboperators}
  all-op-inputs-distributed(op→ aux-result2)
  if not aux-result2 then
    result ← false
  else
    all-op-outputs-receive-dist-1-value(op→ aux-result3)
    if not aux-result3 then
      result ← false
    else
      result ← true
    end if
  end if
else {there are optional suboperators}
  all-op-inputs-distributed(op→ aux-result2)
  if not aux-result2 then
    result ← false
  else
    all-op-outputs-receive-dist-value(op→ aux-result3)
    if not aux-result3 then
      result ← false
    else
      result ← true
    end if
  end if
end if
end
```

```
all-op-outputs-receive-dist-1-value(op→ result):
begin
op-output-arguments(op→ out-args)
for all out-arg such that out-arg∈out-args do
  receives-out-dist(out-arg→ aux-result)
  length(aux-result→ n)
  if n≠1 then
    if n=0 then
      error("No distribution for output of operator")
    else {n>1}
      error("Multiple distributions for output of operator")
    end if
    result ← false
    exit
  end if
end for
result ← true
end

all-op-outputs-receive-dist-value(op→ result):
begin
op-output-arguments(op→ out-args)
for all out-arg such that out-arg∈out-args do
  receives-out-dist(out-arg→ aux-result)
  length(aux-result→ n)
  if n=0 then
    error("No distribution for output of operator")
    result ← false
    exit
  end if
end for
result ← true
end
```

Data flow correctness

```
verify-flow-correctness(op→ result):
begin
all-subops-inputs-receive-value(op→ result)
end
```

```
all-subops-inputs-receive-value(op→ result):
begin
subop-list(op→ subops)
for all subop such that subop∈subops do
  all-subop-inputs-receive(subop→ aux-result)
  if not aux-result then
    result ← false
    exit
  end if
end for
result ← true
end

all-subop-inputs-receive(subop→ result):
begin
op-input-arguments(subop→ in-args)
for all in-arg such that in-arg∈in-args do
  receives-in-dist(in-arg→ aux-result1)
  receives-in-flow(in-arg→ aux-result2)
  if not aux-result1 and not aux-result2 then
    error("No distribution nor flow for input of suboperator")
    result ← false
    exit
  end if
end for
result ← true
end
```

Hierarchical operator correctness

```
verify-compound-op-correctness (op→ result):
begin
ops ← ∅
compound-op-correct(op+ops→ result)
end

compound-op-correct (op+ops→ result):
begin
has-decomposition(op→ aux-result)
if aux-result then
  all-subops-correct(op+ops→ result)
else
  result ← true
end if
end
```

```
all-subops-correct (op+ops→ result):
begin
subop-list(op→ subops)
for all subop such that subop∈subops do
  if subop∈ops then
    error("Suboperator appears already in current definition")
    result ← false
    exit
  else
    ops ← ops ∪ subop
    compound-op-correct(subop+ops→ aux-result)
    if not aux-result then
      error("Incorrect definition of suboperator")
      result ← false
      exit
    end if
  end if
end for
end
```

Adequacy of the initialisation rule base to the unvalued parameters in the operator

```
verify-param-init (op→ result):
begin
has-init-rb(op→ aux-result)
if aux-result then
  verify-init-unvalued-param-completeness(op→ result)
else
  verify-valued-params(op→ result)
end if
end

verify-init-unvalued-param-completeness (op→ result):
begin
op-init-rb(op→ rb)
treats-params(rb→ treated-params)
op-parameters(op→ params)
for all param such that param∈params do
  init(param→ aux-result)
  if aux-result=None and param∉treated-params then
    error("No initialisation rule for unvalued parameter of operator")
    result ← false
    exit
  end if
end for
result ← true
end
```

```
verify-valued-params (op→ result):
begin
op-parameters(op→ params)
for all param such that param∈params do
  init(param→ aux-result)
  if aux-result=None then
    error("Unvalued parameter exists but no initialisation RB in operator")
    result ← false
    exit
  end if
end for
result ← true
end
```

Adequacy of the choice rule base to the suboperators in the specialisation

```
verify-subops-choice (op→ result):
begin
has-choice-rb(op→ aux-result)
if aux-result then
  verify-choice-subop-completeness(op→ result)
else
  error("Specialisation but no choice RB in operator")
  result ← false
end if
end

verify-choice-subop-completeness(op→ result):
begin
op-choice-rb(op→ rb)
treats-subops(rb→ treated-subops)
subop-list(op→ subops)
for all subop such that subop∈subops do
  if subop∉treated-subops then
    error("No choice rule for suboperator in operator")
    result ← false
    exit
  end if
end for
result ← true
end
```

Adequacy of the repair rule base to the problems diagnosed by the evaluation rule base

verify-assess-repair (op→ result):
begin
has-assess-rb(op→ aux-result1)
has-repair-rb(op→ aux-result2)
if aux-result1 **then**
  if aux-result2 **then**
    verify-repair-assess-completeness(op→ result)
  else {there exists an assessment RB but no repair RB}
    verify-assess-continues(op→ result) {OK if repair is not needed}
  end if
else
  if aux-result2 **then** {there is no assessment RB but there exists a repair RB}
    warning("Repair and no assessment implies non-local repair in operator")
    result ← true
  end if
end if
end

verify-repair-assess-completeness (op→ result):
begin
all-failures-are-repaired(op→ result)
end

all-failures-are-repaired (op→ result):
begin
op-assess-rb(op→ rb1)
op-repair-rb(op→ rb2)
diagnoses-failures(rb1→ diagnosed-failures)
treats-failures(rb2→ treated-failures)
for all diagnosed-failure such that diagnosed-failure∈diagnosed-failures **do**
  if diagnosed-failure∉treated-failures **then**
    error("Diagnosed failure exists but no corresponding repair rule in operator")
    result ← false
    exit
  end if
end for
end

verify-assess-continues (op→ result):
begin
op-assess-rb(op→ rb)
diagnoses-failures(rb1→ diagnosed-failures)
if diagnosed-failures<>∅ **then**
  error("Diagnosed failures exist but no repair RB in operator")
  result ← false
else
  result ← true
end if
end

Repair path correctness

verify-repair-path-correctness (op→ result):
begin
op-repair-rb(op→ rb)
treats-failures(rb→ failures)
ops ← ∅
ops ← *cup* op
for all failure such that failure∈failures **do**
  repair-paths-correct(op+failure+ops→ aux-result)
  if not aux-result **then**
    result ← false
    exit
  end if
end for
result ← true
end

```
repair-paths-correct (op+failure+ops→ result):
begin
has-repair-rb(op→ aux-result)
if not aux-result then
   error("Failure might be received but no repair RB in operator")
   result ← false
   exit
end if
op-repair-rb(op→ rb)
translates-failure(rb+failure→ translations)
if translations=∅ then
   error("Failure might be received but no repair rule for it in operator")
   result ← false
   exit
end if
for all translation such that translation∈translations do
   repair-path-correct(op+translation+ops→ aux-result)
   if not aux-result then
      result ← false
      exit
   end if
end for
result ← true
end

repair-path-correct (op+failure-translation+ops→ result):
begin
if failure-translation=re-execute then {failure is solved}
   result ← true
else {failure is not solved}
   all-op-paths-correct(op+failure-translation+ops→ result)
end if
end
```

```
all-op-paths-correct (op+failure-translation+ops→ result):
begin
op-repair-rb(op→ rb)
transmits-failure(rb+failure-translation→ target-ops)
if target-ops=∅ then
  error("Failure is not solved nor transmitted by operator")
  result ← false
  exit
end if
for all target-op such that target-op∈target-ops do
  if target-op∈ops then
    warning("Operator appears already in current repair path")
  end if
  ops-copy ← ops
  ops-copy ← ops-copy ∪ target-op
  repair-paths-correct(target-op+failure-translation+ops-copy→ aux-result)
  if not aux-result then
    error("Incorrect repair path from operator")
    result ← false
    exit
  end if
end for
result ← true
end
```

Adequacy of the adjustment rule base to the treatments in the repair rule base

```
verify-repair-adjust (op→ result):
begin
has-repair-rb(op→ aux-result1)
has-adjust-rb(op→ aux-result2)
if aux-result1 then {there exists a repair RB}
  repair-rb-requires-reexecution(op→ aux-result3)
  if not aux-result2 and aux-result3 then
    error("Re-execution required but no adjustment RB in operator")
    result ← false
  else if aux-result2 and not aux-result3 then
    warning("Adjustment RB exists and re-execution not required in operator")
  end if
else {there is no repair RB}
  if aux-result2 then
    warning("Adjustment RB exists and no repair RB in operator")
  end if
end if
end
```

Redundancy- and conflict-freeness of rule bases

verify-rb-redundancy-and-conflicts (rb→ result):
begin
fill-rb-precondition-and-action-tables(rb→ rb)
check-rb-for-redundancy-and-conflicts(rb→ result)
end

```
fill-rb-precondition-and-action-tables (rb→ rb):
begin
rule-list(rb→ rules)
length(rules→ n)
for all i such that 1 ≤ i < n do
  for all j such that i + 1 ≤ i ≤ n do
    nth(rules+i→ ri)
    nth(rules+j→ rj)
    precs-have-logical-uniqueness(ri+rj→ aux-result1)
    if aux-result1 then
      pr-tab[i, j] ← Exclusive
    else {no logical uniqueness}
      precs-have-same-terms(ri+rj→ aux-result2)
      if aux-result2 then {no logical uniqueness and all terms are common}
        pr-tab[i, j] ← Equivalent
        acts-are-equivalent(ri+rj→ aux-result3)
        if aux-result3 then
          ac-tab[i, j] ← Equivalent
        else
          acts-are-conflicting(ri+rj→ aux-result4)
          if aux-result4 then
            ac-tab[i, j] ← Conflicting
          else
            ac-tab[i, j] ← Different
          end if
        end if
      else {no logical uniqueness and not all terms are common}
        precs-has-same-term-as(ri+rj→ aux-result2)
        precs-has-same-term-as(ri+rj→ aux-result3)
        if aux-result2 or aux-result3 then
          pr-tab[i, j] ← Subsumed
          acts-are-equivalent(ri+rj→ aux-result4)
          if aux-result4 then
            ac-tab[i, j] ← Equivalent
          else
            acts-are-conflicting(ri+rj→ aux-result5)
            if aux-result5 then
              ac-tab[i, j] ← Conflicting
            else
              ac-tab[i, j] ← Different
            end if
          end if
        else
          pr-tab[i, j] ← NonExclusive
        end if
      end if
    end if
  end for
end for
set-precondition-table(rb+pr-tab→ rb)
set-action-table(rb+ac-tab→ rb)
end
```

```
check-rb-for-redundancy-and-conflicts (rb → result):
begin
fill-rb-rule-table(rb→ rb)
check-rb-table-for-redundancy-and-conflicts(rb→ result)
end

fill-rb-rule-table (rb → rb):
begin
precondition-table (rb→ pr-tab)
action-table (rb→ ac-tab)
rule-list(rb→ rules)
length(rules→ n)
for all i such that 1 ≤ i < n do
  for all j such that i + 1 ≤ i ≤ n do
    r-tab[i, j] ← NonMutuallyExclusive
  end for
end for
for all i such that 1 ≤ i < n do
  for all j such that i + 1 ≤ i ≤ n do
    if pr-tab[i, j]=Exclusive then
      r-tab[i, j] ← MutuallyExclusive
    else if pr-tab[i, j]=Equivalent then
      if ac-tab[i, j]=Equivalent then
        r-tab[i, j] ← Redundant
      else if ac-tab[i, j]=Conflicting then
        r-tab[i, j] ← Conflicting
      end if
    else if pr-tab[i, j]=Subsumed then
      if ac-tab[i, j]=Equivalent then
        r-tab[i, j] ← Subsumed
      else if ac-tab[i, j]=Conflicting then
        r-tab[i, j] ← Conflicting
      end if
    end if
  end for
end for
set-rule-table(rb+r-tab→ rb)
end
```

```
check-rb-table-for-redundancy-and-conflicts (rb → result):
begin
r-table (rb→ r-tab)
rule-list(rb→ rules)
length(rules→ n)
for all i such that 1 ≤ i < n do
  for all j such that i + 1 ≤ i ≤ n do
    if r-tab[i, j]=Redundant then
      error("Redundant rule pair detected in RB")
      result ← false
      exit
    else if r-tab[i, j]=Conflicting then
      error("Conflicting rule pair detected in RB")
      result ← false
      exit
    else if r-tab[i, j]=Subsumed then
      error("Subsumed rule pair detected in RB")
      result ← false
      exit
    else {NonMutuallyExclusive rules}
      warning("Ambiguous rule pair detected in RB")
      nth(rules+i→ ri)
      nth(rules+j→ rj)
      search-for-ambiguous-cases(ri+rj→ expression)
      print-expression(expression)
    end if
  end for
end for
result ← true
end

search-for-ambiguous-cases (rule1+rule2→ conjunct):
begin
is-fireable-rule(rule1→ aux-result1)
is-fireable-rule(rule2→ aux-result2)
if aux-result1 or aux-result2 then
  conjunct ← TRUE
else
  conjunct ← ∅
  precondition-list(rule1→ precs1)
  precondition-list(rule2→ precs2)
  add-conjunct(conjunct+precs1→ conjunct)
  add-conjunct(conjunct+precs2→ conjunct)
end if
end
```

Completeness of rule bases

verify-rb-completeness (rb→ result):
begin
check-rb-for-completeness(rb→ aux-result)
if not aux-result **then**
  rule-list(rb→ rules)
  length(rules→ n)
  if n=1 **then**
    print-uncovered-cases(rb)
  else {n>1}
    search-for-completeness-description(rb→ expression)
    print-expression(expression)
  end if
  result ← false
else
  result ← true
end if
end

check-rb-for-completeness (rb→ result):
begin
exist-rb-non-mutually-exclusive-rules(rb→ aux-result1)
rb-only-has-equality(rb→ aux-result2)
rb-has-domains(rb→ aux-result3)
if not aux-result1 **and** aux-result2 **and** aux-result3 **then**
  check-rb-completeness-numerically(rb→ result)
else if aux-result3 **then** {RB domains are defined}
  check-rb-completeness-by-enumeration(rb→ result)
else
  rb-has-fireable-rule(rb→ aux-result4)
  if aux-result4 **then**
    result ← true
  else
    warning("Completeness could not be checked (no domains, no fireable rule)")
    result ← false
  end if
end if
end

```
exist-rb-non-mutually-exclusive-rules (rb→ result):
begin
rule-list (rb→ rules)
rule-table (rb→ r-tab)
length(rules→ n)
for all i such that 1 ≤ i < n do
  for all j such that i + 1 ≤ i ≤ n do
    if r-tab[i, j] <> MutuallyExclusive then
      result ← true
      exit
    end if
  end for
end for
result ← false
end
```

Numerical completeness check

```
check-rb-completeness-numerically (rb→ result):
begin
rule-list (rb→ rules)
length(rules→ n)
domain-list (rb→ domains)
count ← 0
for all i such that 1 ≤ i ≤ n do
  nth(rules+i→ ri)
  count-rule-covered-cases(ri+domains→ aux-result)
  count ← count + aux-result
end for
get-combinations(domains→ aux-result)
if count=aux-result then
  result ← true
else {count≤aux-result}
  error("Domain value combinations exist which are not covered by the RB cases")
  result ← false
end if
end
```

Enumeration completeness check

```
check-rb-completeness-by-enumeration (rb→ result):
begin
domain-list(rb→ domains)
combination ← ∅
check-rb-completeness(rb+combination+domains→ result)
end
```

```
check-rb-completeness (rb+combination+domains→ result):
begin
first(domains→ domain)
is-enum-domain(domain→ aux-result)
domain-element(domain→ elem)
if aux-result then {different values are specified in the domain}
   enum-domain-values(domain→ values)
   check-enum-completeness(rb+elem+values+combination+domains→ result)
else {lower and upper values are specified in the domain}
   interval-domain-values(domain→ lower-value+upper-value)
   check-interval-completeness(rb+elem+lower-value+upper-value+combination+domains→ re-
   sult)
end if
end

check-enum-completeness (rb+elem+values+combination+domains→ result):
begin
for all val such that val∈values do
   new-combination ← combination ∪ (elem, val)
   check-completeness-along-domains(rb+new-combination+domains→ aux-result)
   if not aux-result then
      result ← false
      exit
   end if
end for
result ← true
end

check-completeness-along-domains (rb+combination+domains→ result):
begin
rb-has-fireable-rule-for(rb+combination→ aux-result)
if aux-result then
   result ← true
   exit
else
   rest(domains→ domains) {removes the first domain}
   if domains=∅ then
      error("Incompleteness detected in RB")
      result ← false
      exit
   else
      check-rb-completeness(rb+combination+domains→ result)
   end if
end if
end
```

check-interval-completeness (rb+elem+lower-value+upper-value+combination+domains→ result):
begin
for all val such that lower-value ≤ val ≤ upper-value **do**
  new-combination ← combination
  new-combination ← new-combination ∪ (elem, val)
  check-completeness-along-domains(rb+new-combination+domains→ aux-result)
  if not aux-result **then**
    result ← false
    exit
  end if
end for
result ← true
end

Completeness characterisation with bd-resolution

search-for-completeness-description (rb→ expression):
begin
rule-list(rb→ rules)
rb-has-domains(rb→ aux-result)
if aux-result **then**
  domain-list(rb→ domains)
  unfold-rules(rules+domains→ rules)
  search-for-covered-cases(rules+domains→ expression)
else
  domains ← ∅
  search-for-covered-cases(rules+domains→ expression)
end if
end

```
search-for-covered-cases (rules+domains→ expression):
begin
implied-rules ← ∅
dnf-expression ← ∅
length(rules→ n)
for all i such that 1 ≤ i < n do
  for all j such that i + 1 ≤ i ≤ n do
    nth(rules+i→ ri)
    nth(rules+j→ rj)
    obtain-bd-resolvent(ri+rj+domains→ conjunct)
    if conjunct<>∅ then
      add-dnf-disjunctor(dnf-expression+conjunct→ dnf-expression)
      if ri∉implied-rules then
        implied-rules ← implied-rules ∪ ri
      end if
      if rj∉implied-rules then
        implied-rules ← implied-rules ∪ rj
      end if
    end if
  end for
end for
length(implied-rules→ m)
if m=0 then
  expression ← ∅
  exit
else if m≠n then {0<m<n}
  for all i such that 1 ≤ i ≤ n do
    nth(rules+i→ ri)
    if ri∉implied-rules then
      precondition-list(ri→ conjunct)
      add-dnf-disjunct(dnf-expression+conjunct→ dnf-expression)
    end if
  end for
end if
factorise(dnf-expression→ expression)
end
```

```
obtain-bd-resolvent (rule1+rule2+domains→ conjunct):
begin
precondition-list(rule1→ precs1)
precondition-list(rule2→ precs2)
aux-conjunct ← ∅
precs1-copy ← precs1
precs2-copy ← precs2
length(precs1→ n)
for all i such that 1 ≤ i ≤ n do
  nth(precs1+i→ termi)
  term-appears-negated-in-position(termi→ m)
  if m≠0 then
    delete(precs1-copy+termi→ precs1-copy)
    delete(precs2-copy+termi→ precs2-copy)
    add-conjunct(aux-conjunct+precs1-copy→ aux-conjunct)
    add-conjunct(aux-conjunct+precs2-copy→ aux-conjunct)
    if aux-conjunct=∅ then
      conjunct ← TRUE
    else
      contains-contradiction(aux-conjunct→ aux-result)
      if not aux-result then
        conjunct ← aux-conjunct
        exit
      else
        aux-conjunct ← ∅
      end if
    end if
    precs1-copy ← precs1
    precs2-copy ← precs2
  end if
end for
end
```

Appendix D

Specification, Implementation and Verification of a BLOCKS-based Instruction

Specification

KIV library specifications

```
 elem =
specification
    sorts elem;
    variables e2, e1, e0, e: elem;

    end specification

 nat =
data specification
    nat = 0
        | . +1 (. -1 : nat)
        ;
    variables n: nat;
    order predicates . < . : nat × nat;
end data specification

 set =
generic specification
    parameter elem using nat target
    sorts set;
    constants @ : set;
```

functions

insert	:	elem × set	→	set ;
. \ .	:	set × elem	→	set ;
select	:	set	→	elem ;
#	:	set	→	nat ;

predicates

. ∈ . : elem × set;
. ⊂ . : set × set;

variables s_2, s_1, s_0, s', s: set;

axioms

set **generated by** @, insert;
¬ e_0 ∈ @,
e_0 ∈ insert(e_1, s) ↔ $e_0 = e_1 \vee e_0 \in s$,
$s_1 \subset s_2 \leftrightarrow (\forall\ e_0.e_0 \in s_1 \rightarrow e_0 \in s_2)$,
$s_1 = s_2 \leftrightarrow s_1 \subset s_2 \wedge s_2 \subset s_1$,
$e_0 \in s \setminus e_1 \leftrightarrow e_0 \in s \wedge e_0 \neq e_1$,
$s \neq @ \rightarrow$ select(s) ∈ s,
#(@) = 0,
¬ e_0 ∈ s → #(insert(e_0, s)) = #(s)+1

end generic specification

Basic specifications

ident =
specification
sorts ident;
variables id: ident;

end specification

value =
specification
sorts value;
constants errorvalue : value;
variables va: value;

end specification

par-valued-item = ident, value

par-operator-inf = ident, ident-set

State specification

valued-item =
generic data specification
parameter par-valued-item

vitem = mkvitem (. .vitid : ident, . .vitval : value);
variables vi: vitem;
end generic data specification

basic-state =
actualize set **with** valued-item **by morphism**
set → state, elem → vitem, @ → $@_{st}$, insert → insertst, \ → $\setminus_{st}$, select → selectst, # → $\#_{st}$, ∈ → $\in_{st}$, ⊂ → $\subset_{st}$, s → st, s' → st', s_0 → st_0, s_1 → st_1, s_2 → st_2, e → vi, e_0 → vi_0, e_1 → vi_1, e_2 → vi_2
end actualize

state =
enrich basic-state **with**
predicates is-item-in-state : state × ident;
variables st: state;

axioms

∀ vi, vi_0.vi $\in_{st}$ st ∧ vi_0 $\in_{st}$ st ∧ vi ≠ vi_0 → vi.vitid ≠ vi_0.vitid,
¬ is-item-in-state($@_{st}$, id_0),
is-item-in-state(insertst(vi_0, st), id_0)
↔ id_0 = vi_0.vitid ∨ is-item-in-state(st, id_0)

end enrich

Operator specification

ident-set =
actualize set **with** ident **by morphism**
set → identset, elem → ident, @ → $@_{ids}$, insert → insertids, \ → $\setminus_{ids}$, select → selectids, # → $\#_{ids}$, ∈ → $\in_{ids}$, ⊂ → $\subset_{ids}$, s → ids, s' → ids', s_0 → ids_0, s_1 → ids_1, s_2 → ids_2, e → id, e_0 → id_0, e_1 → id_1, e_2 → id_2
end actualize

operator-inf =
generic data specification
parameter par-operator-inf
opinf = mkopinf (. .operid : ident, . .operfunc : ident, . .operinput : identset, . .operoutput : identset, . .operparam : identset);
variables opi: opinf;
end generic data specification

basic-operator =
data specification
using operator-inf
operator = mkprimop (. .priminf : opinf) **with** isprimop
| mkcompop (. .compinf : opinf, . .dec : decomposition) **with** iscompop
;
decomposition = nildec
| . seq . (carseq : operator, cdrseq : decomposition) **with** isdecseq

;
variables op: operator; dec: decomposition;
end data specification

operator =
enrich basic-operator **with**
predicates indec : operator × decomposition;

axioms

¬ indec(op, nildec),
isprimop(op_0) → (indec(op, op_0 seq dec) ↔ op = op_0 ∨ indec(op, dec)),
iscompop(op_0)
→ (indec(op, op_0 seq dec) ↔ op = op_0 ∨ indec(op, op_0.dec) ∨ indec(op, dec))

end enrich

Specification of program supervision requirements

partial-ps-requirements = state, operator

Specification of execute-operator competence

execute-operator-competence =
enrich partial-ps-requirements **with**
functions execute-operator : operator × state → state ;
variables primitive: operator; ini-state: state;

axioms

isprimop(primitive)
→ (∀ id. id $\in_{ids}$ primitive.priminf.operoutput
→ is-item-in-state(execute-operator(primitive, ini-state), id)),
isprimop(primitive)
→ (∀ id. is-item-in-state(ini-state, id)
→ is-item-in-state(execute-operator(primitive, ini-state), id))

end enrich

Specification of decompose-sequence competence

decompose-sequence-competence =
enrich execute-operator-competence **with**
functions decompose-sequence : operator × state → state ;
variables sequence: operator; fin-state, ini-state: state;

axioms

iscompop(sequence)
→ (decompose-sequence(sequence, ini-state) = fin-state
→ (∀ id.id $\in_{ids}$ sequence.compinf.operoutput → is-item-in-state(fin-state, id)))

end enrich

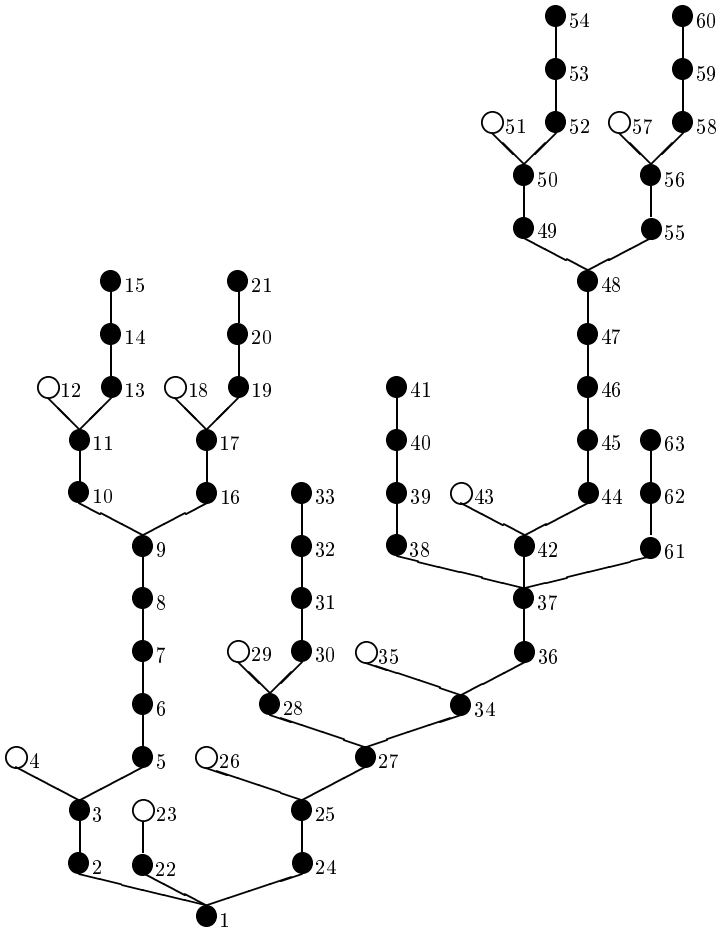
Implementation

Module decompose-sequence-module

```
 decompose-sequence =
module
    export decompose-sequence-competence
    refinement
      representation of operations
          decompose-sequence#   implements   decompose-sequence;

      import execute-operator-competence

      procedures control#(decomposition, state) : state;

      variables out: state;

      implementation


     decompose-sequence#(op, st0; var out)
     begin
       if isprimop(op) then out := st0 else var dec = op.dec in
       control#(dec, st0;out)
     end

     control#(dec, st0; var out)
     begin
       if dec = nildec then out := st0 else
         var op = carseq(dec), st = @st in
         begin
           if isprimop(op) then st := execute-operator(op, st0) else control#(op.dec, st0;st);
           control#(cdrseq(dec), st;out)
         end
     end
```

Verification

Proof of control-iii-1



The goal to prove is:

$\vdash$

isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id)))

Some statistics:

- the tree has 53 nodes
- there were 37 user-interactions
- automation: 30.1 %
- 4 lemmas were used
- There remain no first-order verification conditions
- There remain no other goals

The following lemmas were used:
$\vdash$ isprimop(op) $\vee$ iscompop(op)

$\vdash$
$\langle$control#(dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ op.indec(op, dec) $\wedge$ isprimop(op) $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.priminf.operoutput $\rightarrow$ is-item-in-state(fin-state, id)))

$\vdash$
$\langle$control#(dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ op.indec(op, dec) $\wedge$ iscompop(op) $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id)))

$\vdash$
isprimop(op) $\vee$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ ($\exists$ op_0.indec(op_0, op.dec) $\wedge$ (iscompop(op_0) $\wedge$ id $\in_{ids}$ op_0.compinf.operoutput $\vee$ isprimop(op_0) $\wedge$ id $\in_{ids}$ op_0.priminf.operoutput)))

The following simplifier rules were used: isprimop(mkprimop(opi))
$\neg$ isprimop(mkcompop(opi, dec))
mkcompop(opi, dec).compinf = opi
mkcompop(opi, dec).dec = dec

---

1) **Interactive:** Applied **structural induction** on the following goal

$\vdash$
isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id)))

and got three premises.

---

2) Applied **simplifier** on the goal

---

3) **Interactive:** Applied **insert lemma** compop-output-is-in-dec with the substitution: Θ = {op $\leftarrow$ mkcompop(opi, dec_0)} on the goal
and got two premises.
The first premise is :

⊢
isprimop(op) ∨ (∀ id.id $\in_{ids}$ op.compinf.operoutput → (∃ op_0.indec(op_0, op.dec) ∧ (iscompop(op_0) ∧ id $\in_{ids}$ op_0.compinf.operoutput ∨ isprimop(op_0) ∧ id $\in_{ids}$ op_0.priminf.operoutput)))

The second premise is :

∀ op, ini-state, fin-state.indec(op, dec_0) → isprimop(op) ∨ (⟨control#(op.dec, ini-state;out)⟩out = fin-state → (∀ id.id $\in_{ids}$ op.compinf.operoutput → is-item-in-state(fin-state, id))), ⟨control#(dec_0, ini-state$_0$;out)⟩ out = fin-state$_0$,
id $\in_{ids}$ opi.operoutput, ¬ is-item-in-state(fin-state$_0$, id), isprimop(mkcompop(opi, dec_0)) ∨ (∀ id.id $\in_{ids}$ mkcompop(opi, dec_0).compinf.operoutput → (∃ op_0.indec(op_0, mkcompop(opi, dec_0).dec) ∧ (iscompop(op_0) ∧ id $\in_{ids}$ op_0.compinf.operoutput ∨ isprimop(op_0) ∧ id $\in_{ids}$ op_0.priminf.operoutput)))
⊢

---

5) Applied **pl simplifier** on the following goal

∀ op, ini-state, fin-state.indec(op, dec_0) → isprimop(op) ∨ (⟨control#(op.dec, ini-state;out)⟩out = fin-state → (∀ id.id $\in_{ids}$ op.compinf.operoutput → is-item-in-state(fin-state, id))), ⟨control#(dec_0, ini-state$_0$;out)⟩ out = fin-state$_0$,
id $\in_{ids}$ opi.operoutput, ¬ is-item-in-state(fin-state$_0$, id), isprimop(mkcompop(opi, dec_0)) ∨ (∀ id.id $\in_{ids}$ mkcompop(opi, dec_0).compinf.operoutput → (∃ op_0.indec(op_0, mkcompop(opi, dec_0).dec) ∧ (iscompop(op_0) ∧ id $\in_{ids}$ op_0.compinf.operoutput ∨ isprimop(op_0) ∧ id $\in_{ids}$ op_0.priminf.operoutput)))
⊢

---

6) **Interactive:** Applied **all left** on the goal

---

7) Applied **pl simplifier** on the goal

---

8) **Interactive:** Applied **weakening formulas** on the goal

---

9) **Interactive:** Applied **case distinction** on the goal
and got two premises.

---

10) Applied **simplifier** on the goal

---

11) **Interactive:** Applied **insert lemma** control-results-2 with the substitution: Θ = {ini-state ← ini-state$_0$, dec ← dec_0, fin-state ← fin-state$_0$} on the goal
and got two premises.
The first premise is :

⊢
⟨control#(dec, ini-state;out)⟩out = fin-state → (∀ op.indec(op, dec) ∧ iscompop(op) → (∀ id.id $\in_{ids}$ op.compinf.operoutput → is-item-in-state(fin-state, id)))

The second premise is :

$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(dec_0, $ini\text{-}state_0$;out)$\rangle$ out = $fin\text{-}state_0$,
indec(op, dec_0), id $\in_{ids}$ opi.operoutput, iscompop(op), id $\in_{ids}$ op.compinf.operoutput,
$\neg$ is-item-in-state($fin\text{-}state_0$, id), $\forall$ op.indec(op, dec_0) $\wedge$ iscompop(op) $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state($fin\text{-}state_0$, id))
$\vdash$

---

13) **Interactive:** Applied **all left** on the following goal

$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(dec_0, $ini\text{-}state_0$;out)$\rangle$ out = $fin\text{-}state_0$,
indec(op, dec_0), id $\in_{ids}$ opi.operoutput, iscompop(op), id $\in_{ids}$ op.compinf.operoutput,
$\neg$ is-item-in-state($fin\text{-}state_0$, id), $\forall$ op.indec(op, dec_0) $\wedge$ iscompop(op) $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state($fin\text{-}state_0$, id))
$\vdash$

---

14) Applied **simplifier** on the goal

---

15) Applied **all left** on the goal and closed it!

---

16) Applied **simplifier** on the following goal (from 9)

$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(dec_0, $ini\text{-}state_0$;out)$\rangle$ out = $fin\text{-}state_0$,
indec(op, dec_0), id $\in_{ids}$ opi.operoutput, $\neg$ is-item-in-state($fin\text{-}state_0$, id), isprimop(op) $\wedge$ id $\in_{ids}$ op.priminf.operoutput
$\vdash$

---

17) **Interactive:** Applied **insert lemma** control-results-1 with the substitution: Θ = {ini-state $\leftarrow$ $ini\text{-}state_0$, dec $\leftarrow$ dec_0, fin-state $\leftarrow$ $fin\text{-}state_0$} on the goal
and got two premises.
The first premise is :

$\vdash$
$\langle$control#(dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ op.indec(op, dec) $\wedge$ isprimop(op) $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.priminf.operoutput $\rightarrow$ is-item-in-state(fin-state, id)))

The second premise is :

$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(dec_0, ini-$state_0$;out)$\rangle$ out = fin-$state_0$,
indec(op, dec_0), id $\in_{ids}$ opi.operoutput, isprimop(op), id $\in_{ids}$ op.priminf.operoutput,
$\neg$ is-item-in-state(fin-$state_0$, id), $\forall$ op.indec(op, dec_0) $\wedge$ isprimop(op) $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.priminf.operoutput $\rightarrow$ is-item-in-state(fin-$state_0$, id))
$\vdash$

---

19) **Interactive:** Applied **all left** on the following goal

$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(dec_0, ini-$state_0$;out)$\rangle$ out = fin-$state_0$,
indec(op, dec_0), id $\in_{ids}$ opi.operoutput, isprimop(op), id $\in_{ids}$ op.priminf.operoutput,
$\neg$ is-item-in-state(fin-$state_0$, id), $\forall$ op.indec(op, dec_0) $\wedge$ isprimop(op) $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.priminf.operoutput $\rightarrow$ is-item-in-state(fin-$state_0$, id))
$\vdash$

---

20) Applied **simplifier** on the goal

---

21) Applied **all left** on the goal and closed it!

---

22) **Interactive:** Applied **insert spec-lemma** on the following goal (from 1)

$\langle$control#(op_0.dec, ini-$state_0$;out)$\rangle$ out = fin-$state_0$,
indec(op_0, nildec), $\neg$ isprimop(op_0), $\neg$ $\forall$ id.id $\in_{ids}$ op_0.compinf.operoutput
$\rightarrow$ is-item-in-state(fin-$state_0$, id)
$\vdash$

---

24) Applied **simplifier** on the following goal (from 1)

$\forall$ ini-state, fin-state.isprimop(op_0) $\vee$ ($\langle$control#(op_0.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op_0.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(op_1.dec, ini-$state_0$;out)$\rangle$ out = fin-$state_0$,
$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), indec(op_1, op_0 seq dec_0), $\neg$ isprimop(op_1), $\neg$ $\forall$ id.id $\in_{ids}$ op_1.compinf.operoutput $\rightarrow$ is-item-in-state(fin-$state_0$, id)
$\vdash$

---

25) **Interactive:** Applied **insert lemma** isprimop-or-iscompop with the substitution: $\Theta = \{op \leftarrow op_0\}$ on the goal
and got two premises.
The first premise is :
$\vdash$ isprimop(op) $\vee$ iscompop(op)
The second premise is :

$\forall$ ini-state, fin-state.isprimop(op_0) $\vee$ ($\langle$control#(op_0.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op_0.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(op_1.dec, ini-state$_0$;out)$\rangle$ out = fin-state$_0$,
$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, $\neg$ isprimop(op_1), $\neg$ is-item-in-state(fin-state$_0$, id), isprimop(op_0) $\vee$ iscompop(op_0)
$\vdash$

---

27) **Interactive:** Applied **case distinction** on the following goal

$\forall$ ini-state, fin-state.isprimop(op_0) $\vee$ ($\langle$control#(op_0.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op_0.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(op_1.dec, ini-state$_0$;out)$\rangle$ out = fin-state$_0$,
$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, $\neg$ isprimop(op_1), $\neg$ is-item-in-state(fin-state$_0$, id), isprimop(op_0) $\vee$ iscompop(op_0)
$\vdash$

and got two premises.

---

28) **Interactive:** Applied **insert spec-lemma** on the goal
and got two premises.

---

30) Applied **simplifier** on the following goal

$\forall$ ini-state, fin-state.isprimop(op_0) $\vee$ ($\langle$control#(op_0.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op_0.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(op_1.dec, ini-state$_0$;out)$\rangle$ out = fin-state$_0$,
$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, $\neg$ isprimop(op_1), $\neg$ is-item-in-state(fin-state$_0$, id), isprimop(op_0), isprimop(op_0) $\rightarrow$ (indec(op_1, op_0 seq dec_0) $\leftrightarrow$ op_1 = op_0 $\vee$ indec(op_1, dec_0))
$\vdash$

---

31) **Interactive:** Applied **all left** on the goal

---

32) Applied **simplifier** on the goal

---

33) Applied **all left** on the goal and closed it!

---

34) **Interactive:** Applied **insert spec-lemma** on the following goal (from 27)

∀ ini-state, fin-state.isprimop(op_0) ∨ (⟨control#(op_0.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op_0.compinf.operoutput → is-item-in-state(fin-state, id))), ⟨control#(op_1.dec, ini-$state_0$;out)⟩ out = fin-$state_0$,
∀ op, ini-state, fin-state.indec(op, dec_0) → isprimop(op) ∨ (⟨control#(op.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op.compinf.operoutput → is-item-in-state(fin-state, id))), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, ¬ isprimop(op_1), ¬ is-item-in-state(fin-$state_0$, id), iscompop(op_0)
⊢

and got two premises.

---

36) Applied **pl simplifier** on the following goal

∀ ini-state, fin-state.isprimop(op_0) ∨ (⟨control#(op_0.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op_0.compinf.operoutput → is-item-in-state(fin-state, id))), ⟨control#(op_1.dec, ini-$state_0$;out)⟩ out = fin-$state_0$,
∀ op, ini-state, fin-state.indec(op, dec_0) → isprimop(op) ∨ (⟨control#(op.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op.compinf.operoutput → is-item-in-state(fin-state, id))), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, ¬ isprimop(op_1), ¬ is-item-in-state(fin-$state_0$, id), iscompop(op_0), iscompop(op_0) → (indec(op_1, op_0 seq dec_0) ↔ op_1 = op_0 ∨ indec(op_1, op_0.dec) ∨ indec(op_1, dec_0))
⊢

---

37) **Interactive:** Applied **case distinction** on the goal
and got three premises.

---

38) Applied **simplifier** on the goal

---

39) **Interactive:** Applied **all left** on the goal

---

40) Applied **simplifier** on the goal

---

41) Applied **all left** on the goal and closed it!

---

42) **Interactive:** Applied **insert lemma** compop-output-is-in-dec with the substitution: Θ = {op ← op_1} on the following goal (from 37)

∀ ini-state, fin-state.isprimop(op_0) ∨ (⟨control#(op_0.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op_0.compinf.operoutput → is-item-in-state(fin-state, id))), ⟨control#(op_1.dec, ini-$state_0$;out)⟩ out = fin-$state_0$,
∀ op, ini-state, fin-state.indec(op, dec_0) → isprimop(op) ∨ (⟨control#(op.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op.compinf.operoutput → is-item-in-state(fin-state, id))), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, iscompop(op_0), ¬ is-item-in-state(fin-$state_0$, id), ¬ isprimop(op_1), indec(op_1, op_0.dec)
⊢

and got two premises.
The first premise is :

$\vdash$
isprimop(op) $\vee$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ ($\exists$ op_0.indec(op_0, op.dec) $\wedge$ (iscompop(op_0) $\wedge$ id $\in_{ids}$ op_0.compinf.operoutput $\vee$ isprimop(op_0) $\wedge$ id $\in_{ids}$ op_0.priminf.operoutput)))

The second premise is :

$\forall$ ini-state, fin-state.isprimop(op_0) $\vee$ ($\langle$control#(op_0.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op_0.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(op_1.dec, ini-state$_0$;out)$\rangle$ out = fin-state$_0$,
$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, iscompop(op_0), $\neg$ is-item-in-state(fin-state$_0$, id), $\neg$ isprimop(op_1), indec(op_1, op_0.dec), isprimop(op_1) $\vee$ ($\forall$ id.id $\in_{ids}$ op_1.compinf.operoutput $\rightarrow$ ($\exists$ op_0.indec(op_0, op_1.dec) $\wedge$ (iscompop(op_0) $\wedge$ id $\in_{ids}$ op_0.compinf.operoutput $\vee$ isprimop(op_0) $\wedge$ id $\in_{ids}$ op_0.priminf.operoutput)))
$\vdash$

---

44) Applied **pl simplifier** on the following goal

$\forall$ ini-state, fin-state.isprimop(op_0) $\vee$ ($\langle$control#(op_0.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op_0.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(op_1.dec, ini-state$_0$;out)$\rangle$ out = fin-state$_0$,
$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, iscompop(op_0), $\neg$ is-item-in-state(fin-state$_0$, id), $\neg$ isprimop(op_1), indec(op_1, op_0.dec), isprimop(op_1) $\vee$ ($\forall$ id.id $\in_{ids}$ op_1.compinf.operoutput $\rightarrow$ ($\exists$ op_0.indec(op_0, op_1.dec) $\wedge$ (iscompop(op_0) $\wedge$ id $\in_{ids}$ op_0.compinf.operoutput $\vee$ isprimop(op_0) $\wedge$ id $\in_{ids}$ op_0.priminf.operoutput)))
$\vdash$

---

45) **Interactive:** Applied **all left** on the goal

---

46) Applied **pl simplifier** on the goal

---

47) **Interactive:** Applied **weakening formulas** on the goal

---

48) **Interactive:** Applied **case distinction** on the goal
and got two premises.

---

49) Applied **pl simplifier** on the goal

---

50) **Interactive:** Applied **insert lemma** control-results-2 with the substitution: Θ = {ini-state $\leftarrow$ ini-state$_0$, dec $\leftarrow$ op$_1$.dec, fin-state $\leftarrow$ fin-state$_0$} on the goal
and got two premises.
The first premise is :

$\vdash$
$\langle$control#(dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ op.indec(op, dec) $\wedge$ iscompop(op) $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id)))

The second premise is :

$\forall$ ini-state, fin-state.isprimop(op$_0$) $\vee$ ($\langle$control#(op$_0$.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op$_0$.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(op$_1$.dec, ini-state$_0$;out)$\rangle$ out = fin-state$_0$,
$\forall$ op, ini-state, fin-state.indec(op, dec$_0$) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), iscompop(op), id $\in_{ids}$ op.compinf.operoutput, indec(op, op$_1$.dec), indec(op$_1$, op$_0$ seq dec$_0$), id $\in_{ids}$ op$_1$.compinf.operoutput, iscompop(op$_0$), indec(op$_1$, op$_0$.dec), $\neg$ isprimop(op$_1$),
$\neg$ is-item-in-state(fin-state$_0$, id), $\forall$ op.indec(op, op$_1$.dec) $\wedge$ iscompop(op) $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state$_0$, id))
$\vdash$

---

52) **Interactive:** Applied **all left** on the following goal

$\forall$ ini-state, fin-state.isprimop(op$_0$) $\vee$ ($\langle$control#(op$_0$.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op$_0$.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(op$_1$.dec, ini-state$_0$;out)$\rangle$ out = fin-state$_0$,
$\forall$ op, ini-state, fin-state.indec(op, dec$_0$) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), iscompop(op), id $\in_{ids}$ op.compinf.operoutput, indec(op, op$_1$.dec), indec(op$_1$, op$_0$ seq dec$_0$), id $\in_{ids}$ op$_1$.compinf.operoutput, iscompop(op$_0$), indec(op$_1$, op$_0$.dec), $\neg$ isprimop(op$_1$),
$\neg$ is-item-in-state(fin-state$_0$, id), $\forall$ op.indec(op, op$_1$.dec) $\wedge$ iscompop(op) $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state$_0$, id))
$\vdash$

---

53) Applied **simplifier** on the goal

---

54) Applied **all left** on the goal and closed it!

---

55) Applied **simplifier** on the following goal (from 48)

$\forall$ ini-state, fin-state.isprimop(op$_0$) $\vee$ ($\langle$control#(op$_0$.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op$_0$.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(op$_1$.dec, ini-state$_0$;out)$\rangle$ out = fin-state$_0$,

∀ op, ini-state, fin-state.indec(op, dec_0) → isprimop(op) ∨ (⟨control#(op.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op.compinf.operoutput → is-item-in-state(fin-state, id))), indec(op, op_1.dec), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, iscompop(op_0), indec(op_1, op_0.dec), ¬ is-item-in-state(fin-$state_0$, id), ¬ isprimop(op_1), isprimop(op) ∧ id $\in_{ids}$ op.priminf.operoutput
⊢

---

56) **Interactive:** Applied **insert lemma** control-results-1 with the substitution: Θ = {ini-state ← ini-$state_0$, dec ← op_1.dec, fin-state ← fin-$state_0$} on the goal
and got two premises.
The first premise is :

⊢
⟨control#(dec, ini-state;out)⟩ out = fin-state → (∀ op.indec(op, dec) ∧ isprimop(op) → (∀ id.id $\in_{ids}$ op.priminf.operoutput → is-item-in-state(fin-state, id)))

The second premise is :

∀ ini-state, fin-state.isprimop(op_0) ∨ (⟨control#(op_0.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op_0.compinf.operoutput → is-item-in-state(fin-state, id))), ⟨control#(op_1.dec, ini-$state_0$;out)⟩ out = fin-$state_0$,
∀ op, ini-state, fin-state.indec(op, dec_0) → isprimop(op) ∨ (⟨control#(op.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op.compinf.operoutput → is-item-in-state(fin-state, id))), indec(op, op_1.dec), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, iscompop(op_0), indec(op_1, op_0.dec), isprimop(op), id $\in_{ids}$ op.priminf.operoutput, ¬ is-item-in-state(fin-$state_0$, id), ¬ isprimop(op_1), ∀ op.indec(op, op_1.dec) ∧ isprimop(op) → (∀ id.id $\in_{ids}$ op.priminf.operoutput → is-item-in-state(fin-$state_0$, id))
⊢

---

58) **Interactive:** Applied **all left** on the following goal

∀ ini-state, fin-state.isprimop(op_0) ∨ (⟨control#(op_0.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op_0.compinf.operoutput → is-item-in-state(fin-state, id))), ⟨control#(op_1.dec, ini-$state_0$;out)⟩ out = fin-$state_0$,
∀ op, ini-state, fin-state.indec(op, dec_0) → isprimop(op) ∨ (⟨control#(op.dec, ini-state;out)⟩ out = fin-state → (∀ id.id $\in_{ids}$ op.compinf.operoutput → is-item-in-state(fin-state, id))), indec(op, op_1.dec), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, iscompop(op_0), indec(op_1, op_0.dec), isprimop(op), id $\in_{ids}$ op.priminf.operoutput, ¬ is-item-in-state(fin-$state_0$, id), ¬ isprimop(op_1), ∀ op.indec(op, op_1.dec) ∧ isprimop(op) → (∀ id.id $\in_{ids}$ op.priminf.operoutput → is-item-in-state(fin-$state_0$, id))
⊢

---

59) Applied **simplifier** on the goal

---

60) Applied **all left** on the goal and closed it!

---

61) **Interactive:** Applied **all left** on the following goal (from 37)

$\forall$ ini-state, fin-state.isprimop(op_0) $\vee$ ($\langle$control#(op_0.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op_0.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), $\langle$control#(op_1.dec, ini-state$_0$;out)$\rangle$ out = fin-state$_0$,
$\forall$ op, ini-state, fin-state.indec(op, dec_0) $\rightarrow$ isprimop(op) $\vee$ ($\langle$control#(op.dec, ini-state;out)$\rangle$ out = fin-state $\rightarrow$ ($\forall$ id.id $\in_{ids}$ op.compinf.operoutput $\rightarrow$ is-item-in-state(fin-state, id))), indec(op_1, op_0 seq dec_0), id $\in_{ids}$ op_1.compinf.operoutput, iscompop(op_0), $\neg$ is-item-in-state(fin-state$_0$, id), $\neg$ isprimop(op_1), indec(op_1, dec_0)
$\vdash$

---

62) Applied **simplifier** on the goal

---

63) Applied **all left** on the goal and closed it!

Bibliography

[Ayel, 1988] Ayel, M. (1988). Protocols for Consistency Checking in Expert System Knowledge Bases. In *Proc. Eighth European Conference on Artificial Intelligence (ECAI-88)*, pages 220–225, Munich, Germany.

[Batarekh et al., 1991] Batarekh, A., Preece, A., Bennett, A., and Grogono, P. (1991). Specifying an Expert System. *Expert Systems with Applications*, 2:285–303.

[Beauvieux and Dague, 1988] Beauvieux, A. and Dague, P. (1988). Interactive checking of Knowledge Base consistency. In *International Computer Science Conference*, pages 567–574, Hong Kong.

[Beauvieux and Dague, 1990] Beauvieux, A. and Dague, P. (1990). A General Consistency (Checking and Restoring) Engine for Knowledge Bases. In *Proc. Ninth European Conference on Artificial Intelligence (ECAI-90)*, pages 77–82.

[Benjamins, 1995] Benjamins, R. (1995). Problem-Solving Methods for Diagnosis and their Role in Knowledge Acquisition. *International Journal of Expert Systems – Research and Applications*, 8(2):93–100.

[Benjamins et al., 1996a] Benjamins, R., Fensel, D., and Straatman, R. (1996a). Assumptions of Problem-Solving Methods and their Role in Knowledge Engineering. In *Proc. Twelfth European Conference on Artificial Intelligence (ECAI-96)*.

[Benjamins et al., 1996b] Benjamins, R., Nunes de Barros, L., and Valente, A. (1996b). Constructing Planners Through Problem-Solving Methods. In *Proc. Tenth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW-96)*, Banff, Canada.

[Benjamins and Pierret-Golbreich, 1996] Benjamins, R. and Pierret-Golbreich, C. (1996). Assumptions of Problem-Solving Methods. In Shadbolt, N., O'Hara, K., and Schreiber, G., editors, *Proc. Ninth European Knowledge Acquisition Workshop (EKAW-96)*, number 1076 in Lecture Notes in Artificial Intelligence. Springer-Verlag.

[Bodington, 1995] Bodington, R. (1995). A Software environment for the automatic configuration of inspection systems. In *Proc. First International Workshop on Knowledge-Based Systems for the (re) Use of Program Libraries (KBUP-95)*, pages 99–108, Sophia Antipolis, France. INRIA.

[Chang et al., 1990] Chang, C., Combs, J., and Stachowitz, R. (1990). A Report on the Expert Systems Validation Associate (EVA). *Expert Systems with Applications*, 1:217–230.

[Chien, 1994] Chien, S. (1994). Automated synthesis of image processing procedures for a large-scale image database. In *Proc. First IEEE International Conference on Image Processing (ICIP-94)*, volume 3, pages 796–800, Austin, USA.

[Chien, 1996] Chien, S. (1996). Intelligent Tools for Planning Knowledge Base Development and Verification. In Shadbolt, N., O'Hara, K., and Schreiber, G., editors, *Proc. Ninth European Knowledge Acquisition Workshop (EKAW-96)*, number 1076 in Lecture Notes in Artificial Intelligence, pages 321–337. Springer-Verlag.

[Clément and Thonnat, 1993] Clément, V. and Thonnat, M. (1993). A Knowledge-Based Approach to Integration of Image Processing Procedures. *Computer Vision, Graphics and Image Processing: Image Understanding*, 57(2):166–184.

[Clouard et al., 1993] Clouard, R., Porquet, C., Elmoataz, A., and Revenu, M. (1993). Resolution of image processing problems by dynamic planning within the framework of the blackboard model. In *Proc. SPIE International Symposium*, volume 2056, pages 419–429, Boston, USA.

[Cornelissen et al., 1997] Cornelissen, F., Jonker, C., and Treur, J. (1997). Compositional Verification of Knowledge-Based Systems: a Case study for Diagnostic Reasoning. In *Proc. European Symposium on the Verification and Validation of Knowledge Based Systems (EUROVAV-97)*, pages 129–141, Leuven, Belgium. Katholieke Universiteit Leuven.

[Cragun and Steudel, 1987] Cragun, B. and Steudel, H. (1987). A decision-table-based processor for checking completeness and consistency in rule-based expert systems. *International Journal of Man-Machine Studies*, 26:633–648.

[Crevier, 1993] Crevier, D. (1993). Expert system as design aids for artificial vision systems: a survey. In *Proc. SPIE International Symposium*, volume 2056, pages 84–96, Boston, USA.

[Crubézy, 1999] Crubézy, M. (1999). *Pilotage de programmes pour le traitement d'images médicales*. PhD thesis, Université de Nice-Sophia Antipolis. Submitted.

[Crubézy et al., 1997] Crubézy, M., Aubry, F., Moisan, S., Chameroy, V., Thonnat, M., and Di Paola, R. (1997). Managing Complex Processing of Medical Image Sequences by Program Supervision Techniques. In *Proc. SPIE Medical Imaging (SPIE-MI-97)*, volume 3035, pages 614–625, Newport Beach, USA.

[Crubézy et al., 1998] Crubézy, M., Marcos, M., and Moisan, S. (1998). Experiments in Building Program Supervision Engines from Reusable Components. In *Proc. Workshop on Applications of Ontologies and Problem-Solving Methods, Thirteenth European Conference on Artificial Intelligence (ECAI-98)*, pages 44–53, Brighton, England.

[David et al., 1993] David, J. M., Krivine, J. P., and Simmons, R. (1993). *Second Generation Expert Systems*, chapter Second Generation Expert Systems: A Step Forward in Knowledge Engineering, pages 3–23. Springer-Verlag.

[de Hoog et al., 1994] de Hoog, R., Martil, R., Wielinga, B., Taylor, R., and van de Velde, W. (1994). The Common KADS model set. Technical Report KADS-II/M1/DM1.1b/UvA/018/6.0, University of Amsterdam.

[de Kleer, 1986] de Kleer, J. (1986). An assumption-based TMS. *Artificial Intelligence*, 28:127–161.

[Ehrig and Mahr, 1985] Ehrig, H. and Mahr, B. (1985). *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*. Monographs on Theoretical Computer Science. Springer-Verlag.

[Eriksson and Musen, 1993] Eriksson, H. and Musen, M. (1993). Conceptual models for automatic generation of knowledge-acquisition tools. *The Knowledge Engineering Review*, 8(1):27–47.

[Fensel and Benjamins, 1996] Fensel, D. and Benjamins, R. (1996). Assumptions in Model-based Diagnosis. In *Proc. Tenth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW-96)*, Banff, Canada.

[Fensel and Groenboom, 1997] Fensel, D. and Groenboom, R. (1997). Specifying Knowledge-Based Systems with Reusable Components. In *Proc. Ninth International Conference on Software Engineering & Knowledge Engineering (SEKE-97)*, Madrid, Spain.

[Fensel and Schönegge, 1997] Fensel, D. and Schönegge, A. (1997). Specifying and Verifying Knowledge-Based Systems with KIV. In *Proc. European Symposium on the Verification and Validation of Knowledge Based Systems (EUROVAV-97)*, pages 107–116, Leuven, Belgium. Katholieke Universiteit Leuven.

[Fensel and Schönegge, 1998] Fensel, D. and Schönegge, A. (1998). Inverse Verification of Problem-Solving Methods. *International Journal of Human-Computer Studies*, 49(4):339–362. Special Issue on Problem-Solving Methods.

[Fensel et al., 1996] Fensel, D., Schönegge, A., Groenboom, R., and Wielinga, B. (1996). Specification and Verification of Knowledge-Based Systems. In *Proc. Workshop on Validation, Verification and Refinement of Knowledge-Based Systems, Twelfth European Conference on Artificial Intelligence (ECAI-96)*.

[Fensel et al., 1998] Fensel, D., van Harmelen, F., Reif, W., and ten Teije, A. (1998). Formal support for Development of Knowledge-Based Systems. *Information Technology Management: An International Journal*, 2(4).

[Fuchß 1994] Fuchß T. (1994). Translating E/R-diagrams into Consistent Database Specifications. Technical Report 2/94, Fakultät für Informatik, Universität Karlsruhe, Germany.

[Fuchß et al., 1995] Fuchß T., Reif, W., Schellhorn, G., and Stenzel, K. (1995). Three Selected Case Studies in Verification. In Broy, M. and Jähnichen, S., editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, number 1009 in Lecture Notes in Computer Science. Springer-Verlag.

[Gennari et al., 1994] Gennari, J. H., Tu, S. W., Rothenfluh, T. E., and Musen, M. A. (1994). Mapping Domain to Methods in Support of Reuse. *International Journal of Human-Computer Studies*, 41:399–424.

[Ginsberg, 1988] Ginsberg, A. (1988). Knowlegde-Base Reduction: A New Approach to Checking Knowledge Bases for Inconsistency & Redundancy. In *Proc. Seventh National Conference on Artificial Intelligence (AAAI-88)*, pages 585–589. AAAI-Press, Menlo Park, CA.

[Gong and Kulikowski, 1994] Gong, L. and Kulikowski, C. (1994). VISIPLAN: A Hierarchical Planning Framework for Composing Biomedical Image Analysis Processes. In *Proc. International Conference on Computer Vision and Pattern Recognition*, pages 718–723.

[Gong and Kulikowski, 1995] Gong, L. and Kulikowski, C. (1995). Composition of Image Analysis Processes Through Object-Centered Hierarchical Planning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 17(10):997–1009.

[Gupta, 1993] Gupta, U. (1993). Validation and Verification of Knowledge-Based Systems: A Survey. *Journal of Applied Intelligence*, 3:343–363.

[Kadstool, 1993] Kadstool (1993). *KADSTOOL. Methodological Guide.* Cap Gemini Innovation.

[Laurent, 1992] Laurent, J. P. (1992). Proposals for a Valid Terminology in KBS Validation. In *Proc. Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 829–834, Vienna, Austria. John Wiley & Sons.

[Le-Lisp, 1991] Le-Lisp (1991). *Le-Lisp version 15.25. Manuel de référence.* Institut National de Recherche en Informatique et en Automatique (INRIA), Le Chesnay, France.

[Ligeza, 1997] Ligeza, A. (1997). Logical Analysis of Completeness of Rule-Based Systems with Dual Resolution. In *Proc. European Symposium on the Verification and Validation of Knowledge Based Systems (EUROVAV-97)*, pages 19–29, Leuven, Belgium. Katholieke Universiteit Leuven.

[Loiseau, 1992] Loiseau, S. (1992). Refinement of Knowledge Bases Based on Consistency. In *Proc. Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 845–849, Vienna, Austria. John Wiley & Sons.

[López et al., 1990] López, B., Meseguer, P., and Plaza, E. (1990). Knowledge-Based Systems Validation: A State of the Art. *AI Communications*, 3(2):58–72.

[Marcos et al., 1995] Marcos, M., Moisan, S., and P. del Pobil, A. (1995). Verification and validation of knowledge-based program supervivion systems. In *Proc. IEEE International Conference on Systems, Man and Cybernetics (SMC-95)*, Vancouver, Canada.

[Marcos et al., 1997] Marcos, M., Moisan, S., and P. del Pobil, A. (1997). A Model-Based Approach to the Verification of Program Supervision Systems. In *Proc. Fourth European Symposium on the Validation and Verification of Knowledge Based Systems (EUROVAV-97)*, pages 231–241, Leuven, Belgium. Katholieke Universiteit Leuven.

[Marcos et al., 1998a] Marcos, M., Moisan, S., and P. del Pobil, A. (1998a). Knowledge Modeling of Program Supervision Task. In *Proc. Eleventh International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA-98-AIE)*, number 1415 in Lecture Notes in Artificial Intelligence, pages 124–133. Springer-Verlag, Benicàssim, Spain.

[Marcos et al., 1998b] Marcos, M., Moisan, S., and P. del Pobil, A. (1998b). Knowledge Modeling of Program Supervision Task and its Application to Knowledge Base Verification. *Applied Intelligence.* Accepted for publication.

[Marcus, 1988] Marcus, S., editor (1988). *Automating Knowledge Acquisition for Expert Systems*. Kluwer Academic Publishers, Boston, USA.

[Matsuyama, 1989] Matsuyama, T. (1989). Expert Systems for Image Processing: Knowledge-Based Composition of Image Analysis Processes. *Computer Vision, Graphics and Image Processing*, 48:22–49.

[Mellis and Ruckert, 1989] Mellis, W. and Ruckert, M. (1989). Checking Consistency in Expert Systems. In *Proc. Journées Internationales sur les Systèmes Experts et leurs Applications*, pages 217–229, Avignon, France.

[Meseguer, 1990] Meseguer, P. (1990). A New Method to Checking Rule Bases for Inconsistency: A Petri Net Approach. In *Proc. Ninth European Conference on Artificial Intelligence (ECAI-90)*, pages 437–442.

[Meseguer, 1992] Meseguer, P. (1992). Incremental Verification of Rule-Based Expert Systems. In *Proc. Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 840–844, Vienna, Austria. John Wiley & Sons.

[Meseguer and Plaza, 1992] Meseguer, P. and Plaza, E. (1992). An overview in the VALID project. In *Proc. IFIP-92*, Madrid, Spain.

[Meseguer and Preece, 1994] Meseguer, P. and Preece, A. (1994). Notes of Tutorial on Validation of Knowledge-Based Systems, Eleventh European Conference on Artificial Intelligence (ECAI-94).

[Meseguer and Preece, 1995] Meseguer, P. and Preece, A. (1995). Verification and validation of knowledge-based systems with formal specifications. *The Knowledge Engineering Review*, 10(4):331–343.

[Mili, 1995] Mili, F. (1995). User-Oriented Library Documentation. In *Proc. First International Workshop on Knowledge-Based Systems for the (re) Use of Program Libraries (KBUP-95)*, pages 12–19, Sophia Antipolis, France. INRIA.

[Musen, 1989] Musen, M. (1989). Automated Support for building and Extending Expert Models. *Machine Learning*, 4:347–375.

[Musen, 1992] Musen, M. (1992). Overcoming the limitations of role-limiting methods. *Knowledge Acquisition*, 4:165–170.

[Nazareth and Kennedy, 1993] Nazareth, D. and Kennedy, M. (1993). Knowledge-Based System Verification, Validation and Testing: The evolution of a discipline. *International Journal of Expert Systems – Research and Applications*, 6:143–162.

[Newell, 1982] Newell, A. (1982). The Knowledge Level. *Artificial Intelligence*, 18:87–127.

[Nguyen et al., 1987] Nguyen, T., Perkings, W., Laffey, T., and Pecora, D. (1987). Knowledge Base Verification. *AI Magazine*, 8(2):69–75.

[Nguyen, 1987] Nguyen, T. A. (1987). Verifying Consistency of Production Systems. In *Proc. of the third Conference on Artificial Intelligence Applications*, pages 4–8, Orlando, FL. IEEE Computer Society.

[Nguyen et al., 1985] Nguyen, T. A., Perkins, W. A., Laffey, T. J., and Pecora, D. (1985). Checking an Expert Systems Knowledge Base for Consistency and Completeness. In *Proc. Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 375–378, Menlo Park, CA. Morgan Kaufmann Publishers, San Mateo, CA.

[Nunes de Barros et al., 1997] Nunes de Barros, L., Hendler, J., and Benjamins, R. (1997). Par-KAP: a Knowledge Acquisition Tool for Building Practical Planning Systems. In *Proc. Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1246–1251, Nagoya, Japan.

[Nunes de Barros et al., 1996] Nunes de Barros, L., Valente, A., and Benjamins, R. (1996). Modeling Planning Tasks. In *Proc. Third International Conference on Artificial Intelligence Planning Systems (AIPS-96)*.

[Orsvärn, 1996] Orsvärn, K. (1996). Principles for Libraries of Task Decomposition Methods —Conclusions from a Case-study. In Shadbolt, N., O'Hara, K., and Schreiber, G., editors, *Proc. Ninth European Knowledge Acquisition Workshop (EKAW-96)*, number 1076 in Lecture Notes in Artificial Intelligence, pages 48–65. Springer-Verlag.

[Pipard, 1988] Pipard, E. (1988). Detection d'incohérences et d'incomplétudes dans les bases de règles: le système INDE. In *Proc. Journées Internationales sur les Systèmes Experts et leurs Applications*, pages 15–33.

[Preece et al., 1992] Preece, A. D., Bell, R. D., and Suen, C. Y. (1992). Verifying knowledge-based systems using the COVER Tool. In *Proc. IFIP-92*, Madrid, Spain.

[Preece and Shinghal, 1992] Preece, A. D. and Shinghal, R. (1992). Verifying Knowledge Bases by Anomaly Detection: An Experience Report. In *Proc. Tenth European Conference on Artificial Intelligence (ECAI-92)*, pages 835–839, Vienna, Austria. John Wiley & Sons.

[Preece and Shinghal, 1994] Preece, A. D. and Shinghal, R. (1994). Foundation and Application of Knowledge Base Verification. *International Journal of Intelligent Systems*, 9:683–701.

[Prerau et al., 1993] Prerau, D., Papp, W., Bhatnagar, R., and Weintraub, M. (1993). Verification and Validation of Expert Systems: Experience with four diverse systems. *International Journal of Expert Systems – Research and Applications*, 6:251–269.

[Puerta et al., 1992] Puerta, A. R., Egar, J. W., Tu, S. W., and Musen, M. A. (1992). A multiple-method knowledge-acquisition shell for the automatic generation knowledge-acquisition tools. *Knowledge Acquisition*, 4:171–196.

[Puuronen, 1987] Puuronen, S. (1987). A tabular rule-checking method. In *Proc. Journées Internationales sur les Systèmes Experts et leurs Applications*, pages 257–268.

[Reif, 1995] Reif, W. (1995). The KIV-Approach to Software Verification. In Broy, M. and Jähnichen, S., editors, *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, number 1009 in Lecture Notes in Computer Science. Springer-Verlag.

[Rousset, 1988] Rousset, M. C. (1988). On the Consistency of Knowledge Bases: The COVADIS System. In *Proc. Eighth European Conference on Artificial Intelligence (ECAI-88)*, pages 79–84, Munich, Germany.

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Prentice Hall.

[Runkel and Birmingham, 1993] Runkel, J. and Birmingham, W. (1993). Knowledge-acquisition in the small: building knowledge-acquisition tools from pieces. *Knowledge Acquisition*, 5:221–243.

[Runkel and Birmingham, 1995] Runkel, J. and Birmingham, W. (1995). Analyzing Tasks to Build Reusable Model-Based KA Tools. In *Proc. Ninth Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW-95)*, Banff, Canada.

[Russel and Norvig, 1995] Russel, S. and Norvig, P. (1995). *Artificial Intelligence. A Modern Approach*. Prentice Hall.

[Schreiber et al., 1994] Schreiber, G., Wielinga, B., de Hoog, R., Akkermans, H., and van de Velde, W. (1994). CommonKADS: A Comprehensive Methodology for KBS Development. *IEEE Expert*, 9(6):28–37.

[Shekhar et al., 1995] Shekhar, C., Moisan, S., and Thonnat, M. (1995). Use of a Real-Time Perception Program Supervisor in a Driving Scenario. In *Proc. International Conference on Recent Advances in Mechatronic (ICRAM-95)*, Istambul, Turkey.

[Stenzel, 1993] Stenzel, K. (1993). A Verified Access Control Model. Technical Report 26/93, Fakultät für Informatik, Universität Karlsruhe, Germany.

[Sunro and O'Keefe, 1993] Sunro, L. and O'Keefe, R. (1993). Subsumption anomalies in Hybrid Knowledge Bases. *International Journal of Expert Systems – Research and Applications*, 6:299–320.

[Suwa et al., 1982] Suwa, M., Scott, A. C., and Shortliffe, E. H. (1982). An Approach to Verifying Completeness and Consistency in a Rule-Based Expert System. *AI Magazine*, 3(4):16–21.

[Thonnat et al., 1995] Thonnat, M., Clément, V., and Ossola, J. (1995). Automatic Galaxy Description. *Astrophysical Letters and Communications*, 31:65–72.

[Thonnat et al., 1994] Thonnat, M., Clément, V., and van den Elst, J. (1994). Supervision of Perception Tasks for Autonomous Systems: The OCAPI Approach. *International Journal of Information Science and Technology*, 3(2):140–163. Also in Technical Report 2000 (1993), Institut National de Recherche en Informatique et en Automatique (INRIA).

[Thonnat and Moisan, 1995] Thonnat, M. and Moisan, S. (1995). Knowledge-based systems for program supervision. In *Proc. First International Workshop on Knowledge-Based Systems for the (re) Use of Program Libraries (KBUP-95)*, pages 4–8, Sophia Antipolis, France. INRIA.

[Thonnat et al., 1999] Thonnat, M., Moisan, S., and Crubézy, M. (1999). Experience in Integrating Image Processing Programs. In *International Conference on Vision Systems (ICVS-99)*, Las Palmas, Spain.

[Uschold, 1998] Uschold, M. (1998). Knowledge level modelling: concepts and terminology. *The Knowledge Engineering Review*, 13(1):5–29.

[Valente, 1994] Valente, A. (1994). Knowledge-Level Analysis of Planning Systems. *SIGART Bulletin*, 6(1):33–41.

[van den Elst, 1996] van den Elst, J. (1996). *Knowledge Modelling for Program Supervision in Image Processing*. PhD thesis, Université de Nice-Sophia Antipolis.

[van Harmelen and Aben, 1995] van Harmelen, F. and Aben, M. (1995). Applying rule-base anomalies to KADS inference structures. In *Proc. Workshop on Verification and Validation of Knowledge-Based Systems, Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 35–41, Montreal, Canada.

[van Harmelen and Ten Teije, 1997] van Harmelen, F. and Ten Teije, A. (1997). Validation and Verification of Diagnostic Systems. In *Proc. European Symposium on the Verification and Validation of Knowledge Based Systems (EUROVAV-97)*, Leuven, Belgium. Katholieke Universiteit Leuven.

[Vincent, 1997] Vincent, R. (1997). *Etude des mécanismes de gestion des erreurs dans les systèmes à base de connaisances de pilotage de programmes.* PhD thesis, Université de Nice-Sophia Antipolis.

[Vincent et al., 1996] Vincent, R., Moisan, S., and Thonnat, M. (1996). Une bibliothèque pour des moteurs de pilotage de programmes. Technical Report 3011, Institut National de Recherche en Informatique et en Automatique (INRIA).

[Wielinga et al., 1994] Wielinga, B., Hassan, H., Olsson, O., Orsvärn, K., Schreiber, G., Terpstra, P., van de Velde, W., and Wells, S. (1994). Expertise Model Definition Document. Technical Report KADS-II/M2/UvA/026/5.0, University of Amsterdam.

[Willamowski et al., 1994] Willamowski, J., Chevenet, F., and Jean-Marie, F. (1994). A Development Shell for Cooperative Problem-Solving Environments. *Mathematics and Computers in Simulation*, 36(4-6):361–379.

Verification and Validation of Knowledge-Based Program Supervision Systems

The aim of program supervision is the automation of the different activities involved in the skilled utilisation of a library of programs. To accomplish this task, program supervision systems need a great deal of knowledge on program utilisation, including the situations in which they can be applied, the combinations of programs typically used, etc. Program supervision systems incorporate this expertise in a knowledge-based architecture. Their distinctive characteristics are the variety of knowledge they employ and its representation, which usually includes structured objects and production rules. Despite of its growing importance, little research has concentrated on the verification and validation of systems with the above characteristics. In this thesis we approach the verification and validation of program supervision systems based on knowledge modeling, exploiting the information about the knowledge they require, its organisation, and the precise way in which they use this knowledge during reasoning. This information allows us to identify the properties that knowledge bases should verify in order to adequately serve for program supervision, properties beyond the consistency and completeness of their implementation. In this thesis we present the tools for knowledge base verification developed according to this approach, as well as some experiments in the application of techniques of software verification to program supervision engines with the purpose of identifying the properties we are interested in.

Keywords: Artificial intelligence, knowledge-based systems, program supervision, verification and validation of knowledge-based systems, knowledge modeling.

Verificación y Validación de Sistemas de Supervisión de Programas Basados en el Conocimiento

La supervisión de programas tiene como objetivo la automatización de las distintas actividades implicadas en la utilización especializada de una librería de programas. Para llevar a cabo esta tarea, los sistemas de supervisión de programas necesitan una gran cantidad de conocimiento sobre la utilización de los programas, incluyendo las situaciones en que pueden ser aplicados, las combinaciones de programas habitualmente utilizadas, etc. Los sistemas de supervisión de programas incorporan esta experiencia en una arquitectura basada en el conocimiento. Sus características distintivas son la variedad de conocimiento que emplean y su representación, la cual normalmente incluye objetos estructurados y reglas de producción. A pesar de su importancia creciente, poca investigación se ha dedicado a la verificación y validación de sistemas con las características anteriores. En esta tesis acometemos la verificación y validación de sistemas de supervisión de programas a partir de un modelado del conocimiento, explotando la información sobre el conocimiento que requieren, su organización y la manera precisa en que utilizan este conocimiento durante el razonamiento. Esta información nos permite identificar las propiedades que las bases de conocimiento deben verificar para servir adecuadamente a la supervisión de programas, propiedades mas allá de la consistencia y completitud de su implementación. En esta tesis presentamos las herramientas de verificación de bases de conocimiento desarrolladas de acuerdo con este enfoque, así como algunos experimentos en la aplicación de técnicas de verificación de programas a motores de supervisión de programas con el fin de identificar las propiedades que nos interesan.

Palabras clave: Inteligencia artificial, sistemas basados en el conocimiento, supervisión de programas, verificación y validación de sistemas basados en el conocimiento, modelado del conocimiento.