Chapter 7

Reflective Improvement

Strategic reasoning, in terms of the *KADS* four-layer model, is rarely employed in previous approaches to qualitative reasoning (see chapter 4). However, as discussed in the previous chapter, it is precisely this strategic layer that should contain the knowledge for reasoning *about* the knowledge represented in the other layers. This chapter describes how the strategic knowledge in the model of expertise can be operationalised as a reflective component 'on top of' the artifact. In particular, this chapter discusses ways of improving the problem solving behaviour of *GARP*. These improvements are based on the strategic reasoning found in the protocols of human problem solving which was described in the previous section. In addition, the framework for strategic reasoning is based on the notion of reasoning *about* a problem solver.

7.1 Knowledge about a Problem Solver

Most knowledge engineers, while observing an artifact solving problems, have a good understanding of why the system solves some of the problems easily, and why it takes too much time, or completely fails, on others. In addition, they often know what modifications should be made for improving the problem solving behaviour of the artifact.

An artifact usually has no knowledge about its own problem solving capabilities. It does not know what problem solving task it was developed for or what problems it can solve within the context of that task. This may hamper the usability of the artifact. For instance, when given a problem not related to the domain of application, the artifact should be able to detect, state and explain that it cannot solve that problem. Instead, the system 'enthusiastically' starts solving the problem without knowing that its enterprise is certain to fail.

To improve the state of the art, in the sense of the artifact being more flexible in applying its knowledge, providing explanation of its reasoning process, being clear about its problem solving limitations, and for cooperating with other problem solvers, the artifact needs to be augmented with components that represent knowledge *about* its problem solving potential. These components can then be used to perform *reflective reasoning* about the artifact (cf. [121]), which should result in a more appropriate application of the artifact's problem solving capabilities to the problem in hand.

7.2 Knowledge Level Reflection: a Point of View

Different definitions for reflection have been described by Maes and Nardi (cf. [101]). For our point of view on reflection the distinction between the conceptual model and the design model (see section 3) is important. The conceptual model refers to an implementation independent description of problem solving behaviour. The design model, on the other hand, specifies the computational techniques that constitute the artifact. Crucial for our approach to reflection is that we want the problem solver to reason about its own problem solving potential in terms of the conceptual model (and not in terms of the design model). Our approach therefore differs from what is known as *computational reflection* (cf. [100]), because the *information exchange* between the problem solver and its reflective counterpart is realised in terms of a knowledge level description [109], i.e. the problem solver reasons about the *knowledge* that it uses in its reasoning process. We will refer to this point of view as *knowledge level reflection* (cf. [6; 112]).

Before describing in more detail how this type of reflection can be realised, a number of terms must be defined first. In particular, we have to specify what we mean by reflective specification, reflective system, and reflective behaviour.

7.2.1 Reflective Specification

Reflective specification refers to a conceptual model of the reflective system that is to be built. It describes the model that a knowledge engineer has of such a reflective problem solver.[1] Characteristic for a reflective specification are the structuring principles that the knowledge engineer uses for modelling the knowledge. These structuring principles (for writing down reflective specifications) are visualised in figure 7.1. Typical for a reflective specification are:

- the *model* the knowledge engineer has of the problem solver, and
- the *reflective knowledge* the knowledge engineer has about the problem solver.

Reflective specifications will be further discussed in section 7.3.

7.2.2 Reflective System

From a reflective specification we can develop a reflective system by implementing the reflective specification in a *structure preserving* way. If the structuring principles, as given above for the reflective specification, are imposed upon the architecture of the artifact, then we consider such a system as a reflective system. Crucial in this respect is that the reflective system uses a *model* of the problem solver as an intermediate step for realising the interaction between its 'reflective' and its 'object' problem solving parts. A reflective system must therefore consist of the following aspects (see also figure 7.1):

Object problem solver The problem solver that is the object of reflective reasoning.

[1] Notice that the term knowledge engineer is used in two ways. One, as the person who has knowledge about some problem solver, and two, as the person who constructs a reflective specification of that knowledge.

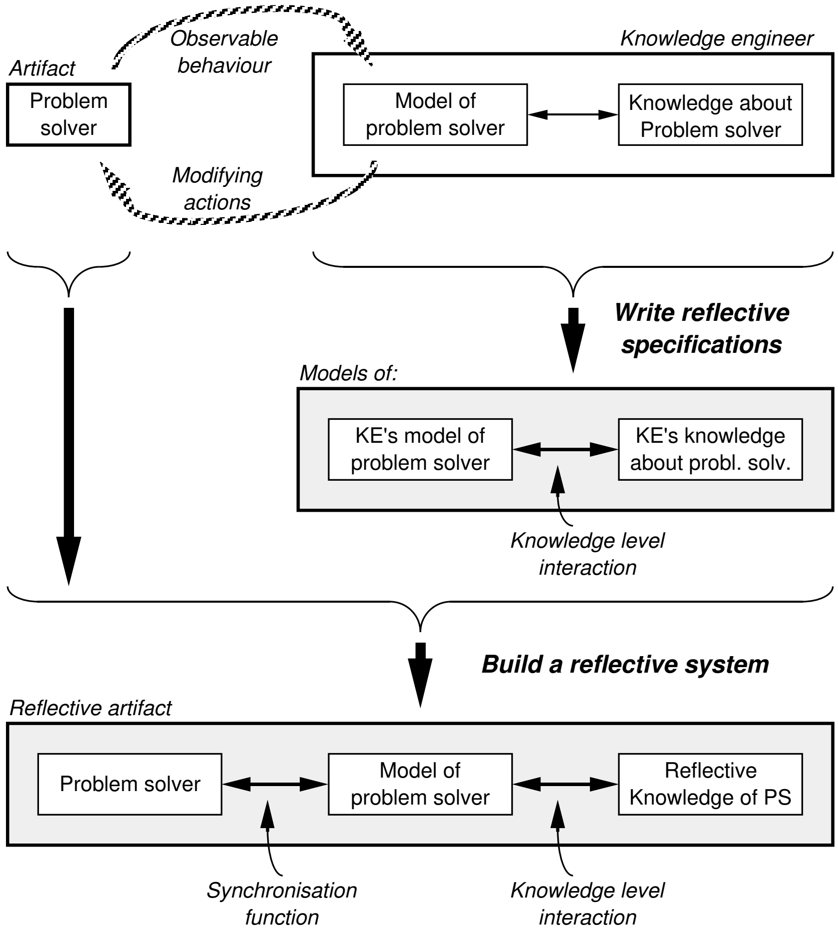

Figure 7.1: Reflective specifications and reflective systems

Model of the object problem solver A knowledge level model that the reflective problem solver has of the object problem solver.

Reflective knowledge Additional knowledge that the reflective problem solver uses for reasoning about the object problem solver.

In order to coordinate the interaction between these three aspects of a reflective system, the following additional entities are required:

Synchronisation function This maps the object problem solver to the model that the reflective system has of that problem solver.

Knowledge level interaction This mechanism specifies how the model of the object problem solver relates to the additional knowledge the reflective system has of the object problem solver.

Reflective architectures and reflective systems are not further discussed in this thesis.

7.2.3 Reflective Behaviour

Reflective behaviour refers to a specific type of problem solving, namely reasoning *about* the problem solving process. The hypothesis here is that there are problem solving behaviours that are not so much concerned with actually solving problems (as knowledge based systems do), but that reason about *how* these problem solvers solve their problems. An example might help to clarify this notion. In the constraint satisfaction community it is generally known that selecting constraints, preceding value assignment, can be optimised by selecting the most constraining first. Reflective behaviour now would be for an artifact to 'sit back' and 'observe' its selection process and at a certain time interrupt the problem solving, having noticed that the selection process is random, and change the selection process according to the optimisation mentioned above.

Reflective behaviour, as defined here, does not require a specific architecture of the system. It refers to a type of reasoning and not to a specific implementation. In figure 7.2 possible combinations of reflective and non-reflective behaviours and systems are given. The figure specifies that, for a system to be reflective, the reflective system (R.S.) must

	Reflective Behaviour	**Non Reflective Behaviour**
Reflective System	• A specific type of architecture (the R.S. has a model of the O.S) *and* • A specific type of reasoning (namely: about problem solving)	• A specific type of architecture (the R.S. has a model of the O.S) *and* • No distinction between specific types of reasoning
Non Reflective System	• No distinction between specific types of architectures *and* • A specific type of reasoning (namely: about problem solving)	• No distinction between specific types of architectures *and* • No distinction between specific types of reasoning

Figure 7.2: Reflective behaviours and reflective systems

reason with a model of the object system (O.S.). For performing reflective behaviour the system must show a specific type of problem solving, namely reasoning about problem solving. Examples of reflective problem solving are given in section 7.5

7.3 Conceptual Model of a Reflective Problem Solver

For developing reflective descriptions of problem solvers we can again use the *KADS* approach for modelling expertise (see chapter 3). Both the knowledge represented in the artifact and the knowledge the engineer has of the artifact, can be modelled with the framework provided by *KADS* (see figure 7.3).[2]

[2]Notice that it is in principle possible to formulate a reflective layer 'on top of' a reflective problem solver. However, instead of allowing this recursion, we impose just one reflective layer 'on top of' an object

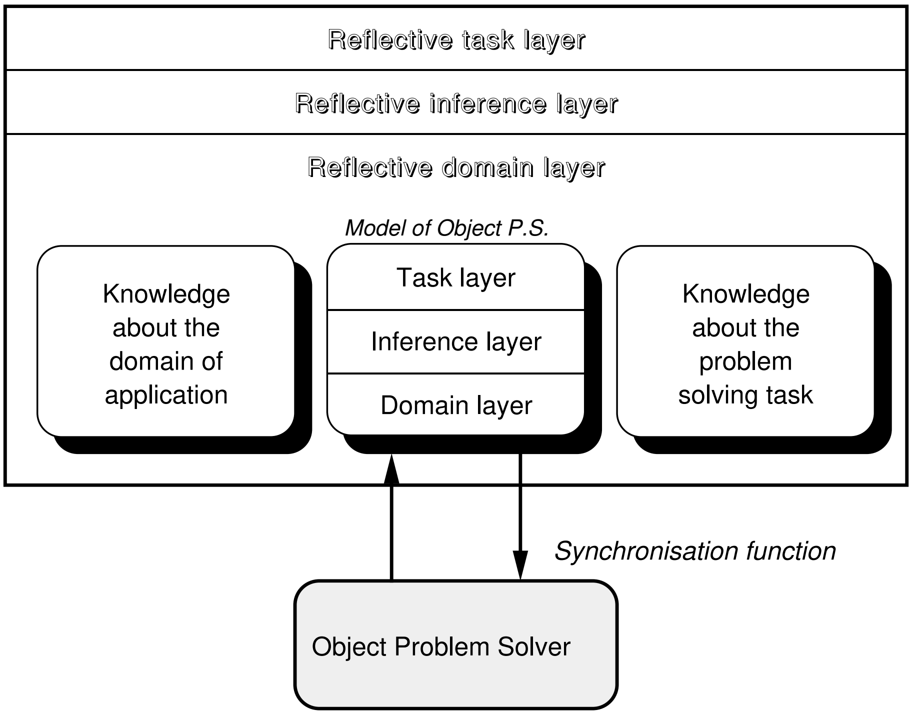

Figure 7.3: Conceptual view of reflective systems

7.3.1 Reflective Domain Layer

At the domain layer of a reflective problem solver the following aspects can be distinguished:

- the model of the object system, and
- additional reflective knowledge, consisting of:
  - knowledge about the problem solving task being performed by the object problem solver, and
  - knowledge about the domain of application.

Particularly relevant is the synchronisation function between the model of the object problem solver and the problem solver itself. This functionality is realised by the *causal connection* [7; 99; 135]. Two aspects are important for this synchronisation: the reflective system must be able to inspect (read) and change (write) the object problem solver. In our approach this interaction between the object and reflective problem solver occurs via the model the reflective system has of the object problem solver.

A difficult problem to solve is the amount of reasoning that the object problem solver may perform before the reflective system interrupts. Different solutions have been proposed [117]. Two opposite views in this respect are:

problem solver.

Eager synchronisation: whenever there is a change in one entity (object problem solver or the model of the object problem solver), it is immediately propagated to the other.

Lazy synchronisation: changes in either entity are not immediately propagated, but only after a certain amount of problem solving has been performed.

The general point is to interact with the object problem solver at the 'right' level of detail. Only the important manipulations carried out by the object problem solver should be inspected, and possibly be modified, by the reflective system. In the case of knowledge level reflection, the conceptual model of the object problem solver must be the basis for determining this level of detail.

7.3.2 Reflective Inference Structure and Task Layer

An inference structure for the reflective problem solver is depicted in figure 7.4. It is taken from the knowledge representation described for the strategy layer in section 3.1.4, and modified such that it applies to *GARP* as an object problem solver. The model of *GARP*,

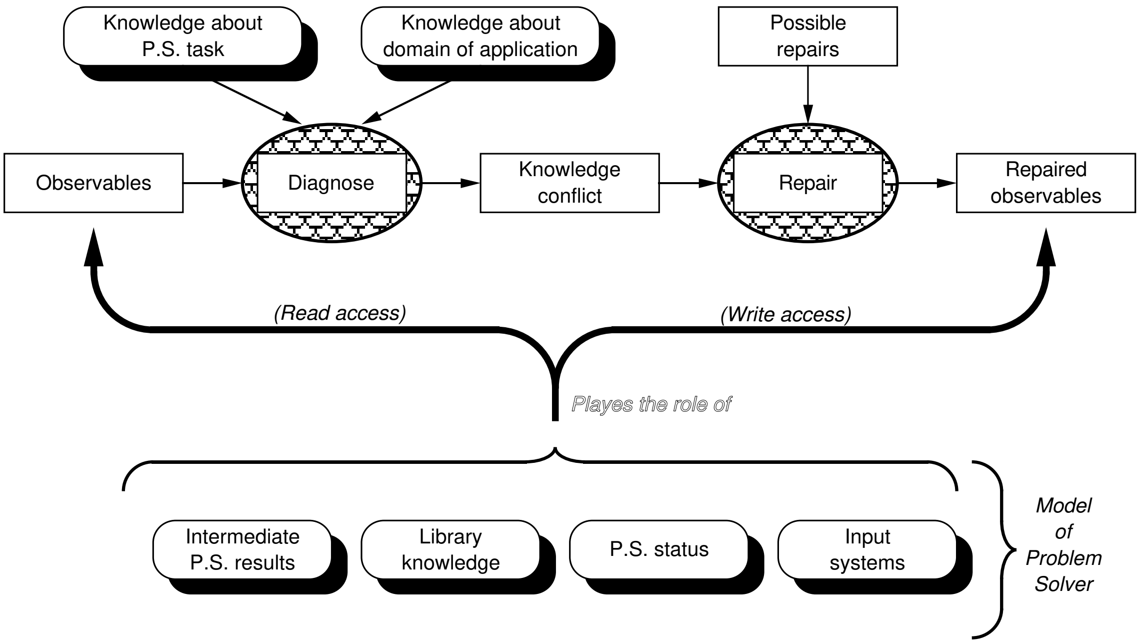

Figure 7.4: Reflective inference structure

at the domain layer, consists of the following parts:

- intermediate problem solving results,
- library knowledge,
- problem solving status, and
- input systems.

The problem solving status refers to what kind of inference is being carried out (recall that each inference maps onto a knowledge source in the inference structure). All the other entities refer to different parts of the knowledge that *GARP* uses for generating behaviour predictions. Each of these entities refers to a number of meta classes which are used by the inferences of the object problem solver.

For the reflective system, each of the above mentioned entities can play the role of *observables*, which are input (=read access) for a diagnostic task.[3] This task in addition uses knowledge about the problem solving task and the domain of application of the object problem solver, and determines whether there is any knowledge in the object problem solver that conflicts with the goal of the problem solving process (=impasse). In order for the object system to realise its problem solving goals, these *knowledge conflicts* have to be repaired. The repair task has a set of *possible repairs* that it can use for this purpose. The output of this task is a *repaired observable*, i.e. one or more modifications (=write access) in the knowledge and/or the reasoning tasks employed by the object problem solver.

The task layer should control the inferences carried out by both the diagnostic and the repair tasks. For the moment, however, we will simplify this notion to executing a diagnostic and a repair step. Further research is needed on this matter.

7.4 Knowledge Conflicts

One of the important meta classes in reflection is the notion of a *knowledge conflict*. It refers to the type of knowledge bottleneck that is dealt with by the reflective reasoning. In a way, reflective reasoning is always concerned with determining to what extent some piece of knowledge from the object problem solver is inadequate (i.e. conflicts with the problem solving goal) and causes the problem solver to behave in an undesired manner. Based on experience with human problem solvers (see chapter 6), we distinguish three types of knowledge conflicts, i.e. three ways in which knowledge represented in the object system can be inadequate with respect to the expected problem solving behaviour.

7.4.1 Inconsistent Knowledge

Knowledge is inconsistent if it contains contradictions. Depending on the scope of the inconsistency, its effect on the problem solving process may range from not proceeding in particular directions, resulting in fewer, possibly less optimal, solutions, to making problem solving impossible altogether. In both cases the problem solving process can only continue in the direction, being blocked by the inconsistency, if the inconsistency is removed.

There is a specific type of inconsistent knowledge that is called *overspecification*. When a constraint satisfaction problem is overspecified it means that there are in principle too many constraints to solve the problem. This is similar to saying that the set of constraints specifying the problem is inherently inconsistent. The only way to solve the inconsistency is by reducing the number of constraints.

[3] Notice, that this realises the knowledge level interaction between the reflective system and the model of the object problem solver.

7.4.2 Missing Knowledge

Missing knowledge refers to knowledge that is needed for the problem solving behaviour, but that is not available to the problem solver. Missing knowledge does not necessarily prevent the reasoning process from continuing. Instead it leaves open certain aspects in the problem solving process which may lead to a larger search space, and probably to more (potential) solutions than wanted, or are actually correct. Remedying missing knowledge can be achieved by *adding* the additional knowledge or by *making assumptions*. More specific assumptions have a stronger effect on curtailing the search space.

A notion used in the area of constraint satisfaction that is a good example of missing knowledge is *underspecification*. When a problem is underspecified it means that there are not enough constraints to ensure solvability in practice. In such cases, too many alternatives can coexist.

We can distinguish between *generally* missing knowledge, i.e. knowledge that is always unknown, and *accidentally* missing knowledge, i.e. knowledge that is unknown for a specific problem solver. Weather forecasting is an example of generally missing knowledge.[4] Arriving at an accurate forecast requires extensive amounts of knowledge. However, essential parts of that knowledge are always unknown, regardless of who is doing the forecast.

Calculating distances between planets can be an example of accidentally missing knowledge. Equations for calculating these distances exist, but a specific problem solver may not know what they are, and therefore not be able to solve these problems. In other words, the problems are solvable, but not for the particular problem solver.

Both types of missing knowledge can be compensated for by using some form of default reasoning. However, generally missing knowledge can never be fully acquired whereas accidentally missing knowledge can. The distinction is therefore relevant, because in the former case it is of no use to search for the missing pieces of knowledge. Instead the reasoning process has to be adapted to the uncertainty introduced by the generally missing knowledge. In the latter case, the problem solver might search for a means for acquiring the missing knowledge.

7.4.3 Irrelevant Knowledge

Irrelevant knowledge can manifest itself in many forms, but the basic idea behind this knowledge conflict is that too much knowledge is taken into account. As a result of irrelevant knowledge the problem solving process takes too much detail into account and may get stuck, because of the complexity, or become incomprehensible because of all the detail. *Abstractions* and *selections* are needed to continue with the problem solving or to transform the knowledge to an understandable format.

There is a specific type of irrelevant knowledge that is called *redundant knowledge*. The two differ in the sense that the latter does not contain additional knowledge, but instead refers to knowledge that is represented more than once in the problem solver. When not identified as such, redundant knowledge may lead to similar problems as irrelevant knowledge.

[4] In [117] this knowledge conflict is sometimes referred to as *uncertain knowledge*.

7.4.4 Exclusivity

Knowledge conflicts are not necessarily mutually exclusive. A piece of knowledge can be part of more than one knowledge conflict. For example, a piece of knowledge can be both *inconsistent* and *irrelevant*.

7.5 Two Types of Reflective Reasoning

Two typical forms of reflective reasoning can be distinguished: competence assessment and competence improvement [136]. Competence assessment has *read* access to one or more parts of the object problem solver, but has no *write* access to any part of the problem solver. Competence improvement, on the other hand, not only has read access to one or more parts of the object problem solver, but also has *write* access to one or more parts of the object problem solver. In terms of the inference structure depicted in figure 7.4, competence assessment is concerned only with the diagnostic step, whereas improvement also requires a repair.

7.5.1 Competence Assessment

Competence assessment refers to the process of judging whether a certain problem solver has the potential for solving a particular problem.[5] The first question to be addressed concerns the intelligibility of the problem:

- does the problem solver understand the problem description?

The question of intelligibility involves two other questions:

1. is the problem *syntactically* intelligible, and
2. is the problem *semantically* intelligible?

For answering these questions the reflective system must have a model of the problem solver that both specifies its syntax and its semantics, and determines whether the problem fits these two. This functionality is realised in GARP by the <*assessment of solvability*> function (see also 5.1.2.3).

If a problem solver understands the problem, the next question is concerned with the solvability itself:

- is the problem solvable by the problem solver?

This question also consists of two other questions:

1. is the problem solvable in principle, and
2. is the problem solvable in practice (i.e. by a specific problem solver)?

[5]Notice that also parts of the library knowledge and/or the intermediate P.S. results, are the object of the competence assessment analysis.

Solvable in practice differs from solvable in principle, because it is more restricted. A problem might be solvable in principle, but a specific problem solver might fail because it lacks an essential reasoning technique or a certain piece of knowledge. Focusing on the latter, the question of solvability depends on:

- the identification of the potential knowledge conflicts, and
- the ability of the problem solver to cope with these knowledge conflicts.

Below a number of tentative rules are given for determining the solvability of a problem, depending on whether the problem solver has the potential for coping with the identified knowledge conflict.

The first rule (7.1) is rather straightforward. It specifies that if there is a knowledge conflict of type *inconsistent*, then the problem is not solvable, neither in practice nor by a specific problem solver.

IF there is a knowledge conflict KC
  AND KC is of type inconsistent knowledge
THEN the problem is not solvable

Table 7.1: Inconsistent knowledge

The second rule (7.2) specifies that if there is a knowledge conflict of type *generally missing*, then the problem solver needs to be equipped with techniques that allow for default reasoning concerning the particular type of knowledge that is missing. If such methods are not available for the problem solver then the problem is unsolvable (for that problem solver), although it might be solvable in principle.

IF there is a knowledge conflict KC
  AND KC is of type generally missing knowledge GMK
  AND the object problem solver has default reasoning methods DRM
  AND the DRM deals with the type of knowledge identified in GMK
THEN the problem is solvable by the object problem solver

Table 7.2: Generally missing knowledge

The third rule (7.3) deals with *accidentally missing* knowledge. This means that the knowledge is available somewhere outside the problem solver and that the problem solver therefore needs to apply knowledge acquisition techniques for obtaining the missing knowledge. If the problem solver does not have these acquisition techniques or if the techniques are not adequate for the type of knowledge that is missing, then the problem is in principle solvable, but not solvable by the specific problem solver.

IF there is a knowledge conflict KC AND KC is of type accidentally missing knowledge AMK AND the object problem solver has knowledge acquisition methods KAM AND the KAM deals with the type of knowledge identified in AMK **THEN** the problem is solvable by the object problem solver

Table 7.3: Accidentally missing knowledge

Finally, an example of a knowledge conflict that has been identified as *irrelevant* (7.4). The effect of irrelevant knowledge depends on its size. If the amount of irrelevant knowledge increases, then the need for abstraction from irrelevant details, and selection between pieces of knowledge, becomes greater. If these methods are not available for the problem solver, then with a certain amount of irrelevant knowledge, the search space becomes too large for solving the problem in practice.

IF there is a knowledge conflict KC AND KC is of type irrelevant knowledge IK AND the amount of IK is (too) large AND the object problem solver has no abstraction methods AM AND the object problem solver has no selection methods SM **THEN** the problem is not solvable by the object problem solver

Table 7.4: Irrelevant knowledge

Having decided that the problem is solvable by the problem solver, the final question concerns the quality of the solution:

- what is the quality of the solution provided by the problem solver?

In more detail this involves questions like:

1. is the solution a correct answer to the problem,
2. is the solution the best solution or are there better ones,
3. what parts of the solution are less reliable (or adequate), and
4. is there something lacking in the solution?

Discussing the quality of the answer is not a part of the research presented here.

7.5.2 Competence Improvement

Competence improvement has read and write access to one or more parts of the object problem solver and takes the competence assessment a step further by suggesting remedies for problems occurring during the problem solving process. Competence improvement very much has the flavour of *controlling* the problem solving process of the object problem solver. Regarded in this way, we can say that competence improvement implements a strategic layer on top of an object problem solver (see also section 3.1.4).

The first problem solving task is *monitoring*. It involves both the observation of the problem solving process, performed by the object problem solver, and the identification of differences between what is expected to be observed and what is actually being observed. Monitoring can be done *data-driven*, i.e. the data determine to a large extent what is being taken into account by the monitoring process, or *model-driven*, i.e. a model (of what is expected) determines what has to be monitored. The monitoring process should provide the reflective component with an 'adequate' description of what is 'going on' in the problem solver. In particular, this task should point out to what extent the object problem solver deviates from its expected problem solving behaviour.

Monitoring is closely related to *diagnosis*: the second basic problem solving task relevant for competence improvement. If the monitoring identifies a discrepancy, it is the task of the diagnostic inference engine to find out what knowledge conflict caused the impasse in the problem solving process.

When the knowledge conflict is identified, the next task is to remedy or repair the impasse. A distinction can be made between *control* and *repair*. They differ in the sense that control does not change the contents of the knowledge in the object problem solver, because it only determines what inference step is to be taken next. Repair, on the other hand, not only determines what inference step is to be taken, but the selected inference step necessarily introduces changes in the body of knowledge in the object problem solver. Examples of repair inferences are abstraction, selection and acquisition of additional knowledge. Repair is, in contrast to control, essentially a non-monotonic action.

7.6 Examples of Reflective Control in *GARP*

In this section we apply the theoretical issues introduced in the previous sections to *GARP*. In particular we will focus on how the problem solving behaviour of *GARP* can be analysed and improved by using the notion of knowledge conflicts.

7.6.1 Inconsistent Knowledge

Conditions that have to be true before some knowledge may be used, can be 'inconsistent' with respect to what they are applied to. This inconsistency simply means that the knowledge cannot be used in this particular context, and does not imply that something is wrong with these conditions. The inconsistency discussed in this section is not concerned with that type of inconsistency, but refers to inconsistencies that should not be present in the first place.

Many forms of inconsistency may occur. Some of them are relatively simple and some are very complex. Some of the more complex cases can be found in the partial behaviour models and in the input systems. Between these modelling primitives inconsistencies may appear in the following cases:

- within the input system
- within a partial model, namely:
  - within the conditions
  - within the consequences
  - between the conditions and the consequences
- between the input system and the consequences of a partial model
- between two partial models, namely:
  - between the consequences of the two partial models
  - between the consequences of one partial model and the conditions of another (already applicable) partial model (in this case the order may be relevant).

During the reasoning process the input system is further specified by adding applicable partial models to it. In the above list the input system can therefore be replaced by system model descriptions, illustrating the inconsistencies that might appear between the system model description, at a certain point in the inference process, and the partial model that is a candidate for being added to that description.

Because the input system, the system model descriptions, and the partial models all incorporate the same type of knowledge (namely about system elements, parameters, parameter values, parameter relations and partial models) the inconsistencies are basically the same for all the items mentioned in the list above. For example, inconsistencies between parameter relations are all based on a contradiction in the set of inequality relations: $greater(P2, zero)$, $greater(P1, P2)$ and $smaller(P1, zero)$. Although this example is rather simple, it is easy to see how complex forms of transitivity, distributed over a number of partial models, can increase the complexity. The main problem in these cases is not so much to identify the inconsistency that appears during the derivation of the transitive closure, but to point out exactly what piece of knowledge caused the inconsistency.

7.6.2 Missing Knowledge

Two typical cases of missing knowledge that are relatively easily spotted and that have a significant effect on the search space are concerned with parameter relations, values and derivatives. If the problem solver cannot determine the truth value of a certain parameter relation, and the relation is not contradictory to the knowledge derived so far, it assumes, after a particular point in the reasoning process, that the relation is true and continues the problem solving process. This may lead to more states of behaviour being predicted than would have been the case if the parameter value had been properly specified. Consider, for example, a system consisting of a closed container filled with a substance. If the temperature of the substance is not specified, the problem solver will assume (i.e. if needed and not contradictory to the other knowledge that is derivable without these assumptions) that it might be either below the freezing point, equal to the freezing point, above the freezing point (etc). Specifying the value of the temperature reduces the number of assumptions that have to be made and consequently the number of states that will be generated.

With respect to the derivative of a parameter it is often the case that because of opposing influences on the parameter the final behaviour of the parameter cannot be determined unambiguously. The problem solver then assumes that all changes (increasing, decreasing and steady) are possible, which results in three states of behaviour. The effect of this ambiguity increases significantly when there is more than one independent parameter with an undetermined derivative. For example in case of three ambiguous parameter derivatives 27 (3x3x3) states of behaviour are generated, all of them introducing a new path of behaviour.

Other cases of missing knowledge are much harder to identify. Take for example a missing partial model. It is almost impossible for the problem solver to spot such missing knowledge, although some intuitive heuristics can be used (for example, there should be a partial model for each system element from the isa-hierarchy).

7.6.3 Irrelevant Knowledge

If knowledge is represented too specifically, it will result in a behaviour prediction with more detail than is required, i.e. too many system model descriptions are produced, or too much knowledge is placed within each system model description (the latter is more likely).

Below, three forms of irrelevant knowledge are discussed. In each example, the problem that has to be dealt with is choosing the right level of abstraction to represent and reason about a system. If too much detail is taken into account the problem solving process gets unnecessarily complex and abstractions are needed to simplify the reasoning process.

7.6.3.1 Irrelevant Structural Detail

The configuration of system elements representing a system can be very complex because of all the detail taken into account. During the reasoning process the whole configuration has to be analysed for each state of behaviour which may result in an unnecessarily large amount of detail. Although the reasoning is in principle correct, it takes more time than strictly needed and it makes the interpretation of the output for the user more difficult. This is particularly true when the configuration of system elements does not change during the simulation (see also chapter 6).

7.6.3.2 Removing Irrelevant Parameters

If too many parameters are used for describing the behaviour of some system, there is a danger that more terminations are found by the problem solver than wanted. This problem particularly occurs when the parameters represent similar knowledge about the system in a different way and are insufficiently related to each other (i.e. they seem independent, but are in fact related). In addition, a large set of parameter values makes it hard to interpret the behaviour specified within a specific system model description.

In general parameter redundancy may appear in two forms:

- Two (or more) parameters are erroneously used for describing the same behaviour property (this is essentially a modelling error).
- Two (or more) parameters are used for describing different properties, but the properties represent corresponding behaviour.

With respect to the latter form of redundancy, consider the following example. In the domain of heart diseases the $amount\_of$ oxygen contained by the blood represents a different parameter from the $amount\_of$ blood itself. However, after assuming that the $amount\_of$ oxygen in the blood stays constant, the parameters represent, from a qualitative point of view, the same information. One of the parameters can therefore be removed.

The following algorithm can be used for finding and removing a certain class of irrelevant parameter relations:

1. If for each SMD from the behaviour description, there exists:
   - a parameter (=Par1) in the list of parameters.
   - a parameter (=Par2) in the list of parameters, where Par1 $\neg$ Par2.
   - a *correspondence* relation between Par1 and Par2 in the list of parameter relations.
   - a *d_equal* relation between Par1 and Par2 in the list of parameter relations.

   then Par1 and Par2 together represent irrelevant knowledge in the behaviour description.

2. For each pair of irrelevant parameters (Par1 and Par2) in each SMD from the behaviour description, *act as follows:*
   - *Remove* all the relations between Par1 and Par2 from the list of parameter relations (also within the partial models).
   - *Assign* (randomly) either Par1 or Par2 as the remaining (=Remain) or to be removed (=Remove) parameter.
   - *Remove* the parameter value of Remove from the list of parameter values (also within the partial models).
   - *Remove* the parameter Remove from the list of parameters (also within the partial models).
   - *Replace* all parameter relations in the list of parameter relations that use parameter Remove by similar relations but now using parameter Remain (also within the partial models).
   - *Remove* all duplicated parameter relations from the list of parameter relations (also within the partial models).

7.6.3.3 Removing Irrelevant States of Behaviour

There is a danger of predicting system model descriptions that contain different values for some parameter, but that do not represent different states of behaviour. This happens when a quantity space for a parameter has values that do not correspond to some partial behaviour model in the library. In the domain of the heart diseases (cf. [16]) it turned out that *GARP* found a number of angina pectoris states, all containing the same partial models, but with different values for a parameter.

Angina pectoris refers to a state of behaviour in which the heart has a higher level of activation than normal and the patient perceives pain in the chest because insufficient oxygen is provided by the blood. The conditions for angina pectoris are therefore that the use_of oxygen is greater than the $amount\_of$ oxygen present:

$$greater(Use\_of\_O2, Amount\_O2)$$

and that the use_of oxygen must be higher than normal:

$$greater(Use\_of\_O2, normal(Use\_of\_O2))$$

which refers to the person having a higher activation level than normal. The following quantity space was used for both the $amount\_of$ and the use_of oxygen:

$$quantity\_space(zlnh, X, [point(zero), low, point(normal(X)), high]).$$

Given these definitions it is possible that, when the Use_of_O2 is $high$, the value for the $Amount\_O2$ varies between low, $normal$, and $high$.[6] This means that there are three possible states of behaviour that satisfy the angina pectoris conditions. However, the three states of behaviour did not differ on their partial behaviour models. In other words, below a certain point in the quantity space, the values of the parameters did not introduce any new behaviour and may just as well be left out. A quantity space with less detail would still predict the *relevant* states of behaviour, but with fewer system model descriptions.

There are two possible repairs for this problem, either defining a quantity space with fewer values, or merging the predicted states of behaviour that do not differ on their partial models and replace the different values for the parameter by a parameter relation. In the example above, this relation would be:

$$greater(Amount\_O2, zero)$$

The following algorithm can be used for removing irrelevant system model descriptions:[7]

1. *Find* all SMD's that have the same set of:
    - (a) system elements
    - (b) parameters
    - (c) parameter relations

[6] Notice that $high$ is an interval and that parameters can be 'unequal' in the same interval (see section 4.2.1.5).

[7] If the derivatives of the parameters whose values are being substituted by parameter relations are different, then this information will be lost after applying this algorithm.

   (d) partial behaviour models

   In words: find all the SMD's that only differ on their parameter values $\Rightarrow$ Set of irrelevant SMD's.

2. *Create* a new SMD (=NewSMD)
3. *Select* a specific SMD from the Set of irrelevant SMD's (=CopySMD)
4. *Copy* from CopySMD into NewSMD:
   (a) system elements
   (b) parameters
   (c) parameter relations
   (d) partial behaviour models
5. For each parameter (=Par) from each SMD in the Set of irrelevant SMD's with a parameter value, *act as follows:*
   (a) *If* the parameter value is similar in each SMD
   then copy the parameter value (once) into the NewSMD
   (b) *If* the parameter value differs in two or more SMD's
   then
      1a *Find* the lowest parameter value (=LowestValue)
      1b *Create* a parameter relation for the parameter:
      $greater\_or\_equal(Par, LowestValue)$
      1c *Add* the created parameter relation to the NewSMD
      2a *Find* the highest parameter value (=HighestValue)
      2b *Create* a parameter relation for the parameter:
      $smaller\_or\_equal(Par, HighestValue)$
      2c *add* the created parameter relation to the NewSMD
      3a *Remove* the parameter value in each of the partial behaviour models in the NewSMD
6. For each $from\_relation$ in each SMD in the Set of irrelevant SMD's, *rewrite* this relation as a $from\_relation$ for the NewSMD.
7. For each $to\_relation$ in each SMD in the Set of irrelevant SMD's, *rewrite* this relation as a $to\_relation$ for the NewSMD.

7.7 Concluding Remarks

In this chapter we have described a framework for knowledge level reflection. Based on this, we gave a classification of reflective behaviour and investigated a knowledge level theory of reflection. In particular, we analysed how the notion of knowledge conflicts can be used as a means for reasoning about the competence assessment and improvement of knowledge based systems. It turned out that impasses in the problem solving process of object problem solvers can be identified and described with one of the three basic knowledge conflicts. In addition we discussed how remedies can be used to aid competence improvement.

Although some typical examples have been tried out in experiments (cf. [5]), further research is needed for realising the presented ideas in an implemented reflective problem solver.