1 Introduction

The property of *fitness-for-purpose* states whether running a software component with a supplied set of inputs can satisfy a given goal. Fitness-for-purpose is related to the notion of competence (Wielinga, Akkermans & Schreiber, 1998), but with some important differences. Firstly, competence describes the *general* input-output relationship of a problem-solver, whereas the verification of fitness-for-purpose involves a test of the problem solver and a *specific* set of input instances against a *given* goal. Secondly, fitness-for-purpose is a *descriptive* notion, because it is applied to an existing system, whereas competence is a *prescriptive* term, applied to a system that is to be built.

Descriptions of fitness-for-purpose can enable human- or machine agents to assess the suitability of application of the described component for a particular task. Some established approaches to assessing whether a software component is suitable for solving the task at hand are by *design*, *testing*, *verification & validation*, *proving properties*, and *syntactic/structural matching with requirements*. *Suitability by design* aims to *develop* new components to satisfy requirements. *Suitability by testing* aims to detect faults in existing components by careful preparation of test case inputs. *Suitability by verification & validation* aims to check whether (knowledge-based) systems meet their users' requirements by, for example, identifying redundant or conflicting rules in a knowledge base. *Suitability by proving properties* is potentially more informative. However, when done by hand, it is a difficult and involved process and only feasible for small programs. When automated, it can also be problematic; for example, MacKenzie notes that "wholly automatic theorem provers have so far been considered inadequate to these tasks" (MacKenzie, 1995).

The (semi-)automatic assessment of fitness-for-purpose is becoming increasingly important for two main reasons. Firstly, as the range and sophistication of software increases, it becomes difficult for a human user to make an informed choice of the best software solution for any particular task. The same point applies equally to domain independent reasoning components, such as the problem solving methods (PSMs) (Benjamins & Fensel, 1998). We believe that novice users of such components, in particular, could benefit greatly from advice generated from a fitness-for-purpose analysis. Secondly, we observe a demand for software brokers, which, given some software requirements, return either the software itself, or a reference to such software. In the knowledge acquisition community, the IBROW3 project (Benjamins et al., 1998) intends to build such a broker for the distribution of problem solving components.

Recent work on the description of competence includes that of Benjamins et al. in the context of the IBROW3 project (Benjamins et al., 1998; Benjamins et al., 1999), Fensel et al., as part of an ongoing investigation into the role of *assumptions* (Fensel & Schönegge, 1999; Fensel & Schönegge, 1998), and Wielinga et al., as a methodological approach to the operationalisation of knowledge-level specifications (Wielinga, Akkermans & Schreiber, 1998). We consider Fensel's approach to be the nearest to our work, because it investigates the *context dependency* of an existing problem solver/PSM through the discovery of assumptions. Our work also investigates the context dependency of problem solvers, but through the question of task suitability with the *available* knowledge. Thus, whilst Fensel investigates a

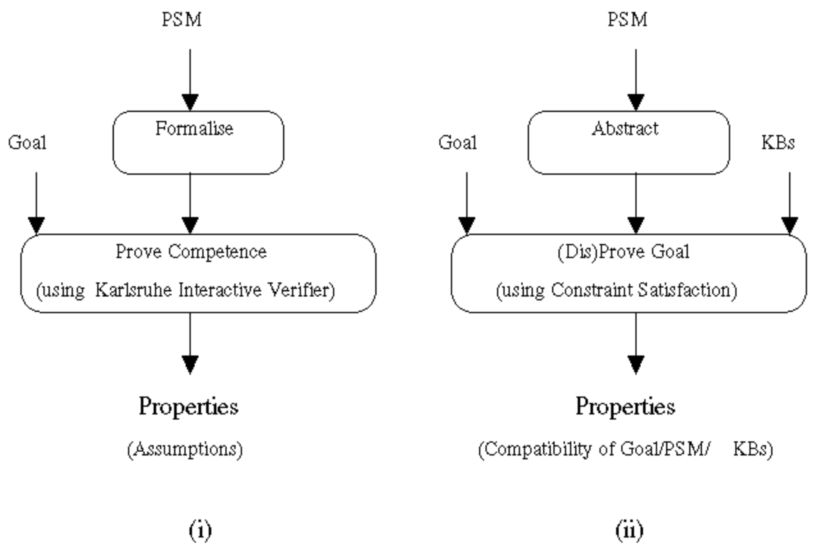

Figure 1: Comparison of the Fensel approach (i), and our approach (ii), to the discovery of problem solving properties

problem solving method in isolation of its inputs in order to *derive* suitability conditions, we take a problem solver together with its inputs and *test* their combined suitability for solving a specific task. Both lines of inquiry are intractable, and therefore demand some compromise(s) in any implementation. Fensel's compromise concerns the level of automation of his proof mechanism, which is an *interactive* verifier, rather than a fully automated proof mechanism. Since we would like to generate advice at *run-time* for the potential user of a problem solver, we compromise instead with the deductive power of our proof procedure. The advantages of our approach lie in the ability to combine the results of multiple problem solvers (through propagation mechanisms), and to run as a batch process. Figure 1 compares the process of assumption hunting of Fensel et al., with our approach to matching a problem solver to a toolkit user's goal.

Our approach to fitness-for-purpose analysis was developed to assist in the generation of advice for novice users of knowledge acquisition tools (KA tools) and problem-solvers in the MUSKRAT system (Graner & Sleeman, 1993; Sleeman & White, 1997; White & Sleeman, 1998; White, 2000). MUSKRAT is a **MU**lti **S**trategy **K**nowledge **R**efinement and **A**cquisition **T**oolbox that makes it easier for novice users to apply and combine the incorporated software tools to solve some specific task. When generating advice on the application of a chosen problem-solver, MUSKRAT should be able to differentiate between the following three cases for the knowledge bases available to the problem-solver.

Case 1: The problem-solver can be applied with the knowledge bases already available, i.e., no acquisition or modification of knowledge is necessary.

Case 2: The problem-solver needs knowledge base(s) that are not currently available; therefore these knowledge base(s) must first be acquired.

Case 3: The problem-solver needs knowledge base(s) that are not currently available, but the requirements can be met by modifying existing knowledge base(s).

The phrase "fitness-for-purpose" and the computational approximation to it, as described in this paper, represents our approach to recognising case 1. Cases 2 and 3 are interesting research topics in their own right and are also being investigated. For case 2, see White & Sleeman (2001); for case 3, see Hameed, Sleeman & Preece (2001).

In the next section, we define more exactly what we mean by fitness-for-purpose, and explain the computational difficulties which arise in its analysis. In section 3, we describe how we approach these difficulties by considering an *approximation* of fitness-for-purpose, rather than the *actual* fitness-for-purpose. In section 4, we explain how we are implementing these ideas using constraint programming in Common LISP. Finally, in section 5 we summarise the ideas, relate them to other work in the field, and indicate some possible directions for future work.

2 Fitness-for-Purpose

When the advisory subsystem of MUSKRAT addresses the problem of fitness-for-purpose, it is in effect posing the question "Is it possible to solve the given task instance with the available problem-solver [2], knowledge bases, and KA tools?". Clearly, this is a very difficult question to answer without actually running the problem-solver, regardless of whether the answer is generated by a human or a machine. Indeed, the theory of computation has shown that it not possible, in general, to determine whether a program (in this case, a problem-solver) terminates. This is the *Halting Problem* (Turing, 1937). Therefore the only way to affirm the suitability of a problem-solver for solving a particular task is to run it and test the outcome against the goal. For the purposes of generating advice in a multistrategy toolbox, however, we cannot afford this luxury, particularly since running the problem-solver could often be computationally intensive. We prefer instead to pose the weaker question "Is it *plausible* that the given task instance could be solved with the available problem-solver, knowledge bases, and KA tools?". We have shown that it is possible to demonstrate computationally that some configurations of a task, problem-solver, existing knowledge bases and KA tools cannot, even *in principle*, generate an acceptable solution to the task. Such situations form a set of *recognisably implausible* configurations with respect to the problem at hand. Furthermore, the computational cost associated with recognising this implausibility is, in many cases, far less than that associated with running the problem-solver.

For example, consider a question from simple arithmetic: is it true that $22 \times 31 + 11 \times 17 + 13 \times 19 = 1097$? Rather than evaluate the left hand side of the equation straight away, let us first inspect the problem to see if it is reasonable. We know that when an even number is multiplied by an odd number, the result is always even; and that an odd number multiplied by an odd number is always odd. Therefore the left hand side of the equation could be rewritten as $< even > + < odd > + < odd >$. Likewise, an even number added to an odd number is always odd, and the sum of two odd numbers is always even. Then evaluating left to right, we have $< odd > + < odd >$, which is $< even >$. Since 1097 is not even, it cannot be the result of the evaluation. We have thus answered the question without having to do the "hard" work of evaluating the actual value of the left hand side.

As another example, consider the truth of the statement "If **P**igs can fly, then I'm the **Q**ueen of Sheba", which we write as P $\Rightarrow$ Q. Given that the premise P is false, we can use the truth table for logical implication [3] to derive that the whole statement is true, since any implication with a false premise is true. Notice that we derived our result *without having to know* the truth of the consequent Q. In a similar way, it is possible to investigate the outputs of programs (in particular, problem-solvers) without needing complete information about their inputs. This issue becomes important if running a problem solver has a high cost associated with it, such as the time it takes to perform an intensive search [4] . In such cases, a preliminary investigation of the plausibility of the task at hand could save much time if the intended problem solver configuration can be shown to be unsuitable for the task. We consider such an investigation to be a kind of "plausibility test" that should be carried out before running the actual problem solver. The idea was suggested as part of a general problem solving framework by Polya (Polya, 1957). In his book "How to Solve It", he proposed some general heuristics for tackling problems of a mathematical nature, including an initial inspection of the problem to understand the condition that must be satisfied by its solution. For example, is the condition sufficient to determine the problem's "unknown"? And is there redundancy or contradiction in the condition? Polya summarised the issue by asking the following:

> Is our problem *reasonable*? This question is useful at an early stage of our work *if* we can answer it easily. If the answer is difficult to obtain, the trouble we have in obtaining it may outweigh the gain in interest.
> *(Polya, 1957)* [5].

Note that the arithmetic example given above first abstracts the problem instance to a different "space" (i.e., from integers to that of odd and even numbers), to which a simpler algebra can be applied. Much of the problem instance detail has been ignored to keep the plausibility test simple. On the other hand, enough detail has been retained for the test to reflect the current problem instance and for it to be useful. In this sense, the plausibility test has *approximated* the task. In the next section, we define a more precise notion of plausibility approximation, and explain how it can be applied to problem-solvers.

3 The Plausibility Approximation to Fitness-for-Purpose

Consider the output of a problem solver applied to a given set of knowledge-base inputs. Our approach compares our knowledge of this output for consistency with the properties we desire (our problem-solving goal). We classify a problem solver's proposed output value as one of *implausible*, *plausible*, *possible* or *actual.* A value that is *implausible* will never be the output of the problem solver. Informally, a *plausible value* cannot be ruled out as a solution through reasoning such as that in the arithmetic or logical examples above [6]. A *possible* value can be a solution to the problem in some cases. An *actual* value is the solution in a given case. It is worth noting that all *possible* values are also *plausible*, and any *actual* value is always *possible*, and hence also *plausible* (see Figure 2).

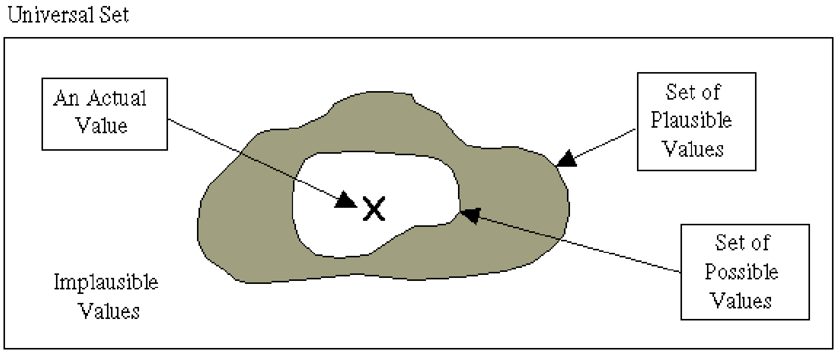

Figure 2: Venn Diagram showing the relationships between plausible, possible, and actual values

As an example of this classification, let us reconsider the arithmetic example given earlier. We reasoned that a *plausible set* for the arithmetic expression $22 \times 31 + 11 \times 17 + 13 \times 19$ is the set of even numbers. So any odd number, such as 1097, is *implausible.* The *possible set*, in this case, has only one value, the *actual value*, 1116. Note that in general the possible set may have many values; for example, if the problem solver employs a stochastic procedure, or uses inputs that are not determined until run-time.

The set of plausible values is an approximation to the set of possible values, but we define it such that *the set of possible values is always contained within the set of plausible values.* This guarantees that an implausible value is not possible, so we can approximate the test for a possible value with a test for plausibility. Note that a plausible value may not necessarily be possible – but it may be hard to prove that it is not possible. Note also that for any gain in computational effort, testing an element for membership of the set of plausible values must be in some sense less expensive than testing it for membership of the set of possible values.

The *hash-code* is a well-known plausibility approximation used in mainstream computer science (see, for example, Brassard & Bratley, 1996). If there is a mismatch between the hash-codes of two objects, then the two objects are (certainly) different. If the hash-codes are the same (they "collide"), then the two objects are plausibly equal. Most importantly, comparing objects' hash-codes is computationally cheaper than comparing the objects themselves.

3.1 Inferences Using the Plausibility Approximation

Our approach recognises that a problem solving task can only be solved if there is at least one member of the possible set of problem-solver outputs that is also a solution to the task. Let us denote the set of acceptable output values (i.e., those that solve the task) by **goal**. If the problem solver *PS* uses the inputs $I = \{I_1, I_2, ...I_n\}$ in roles $R = \{R_1, R_2, ...R_n\}$, then we denote [7] the set of possible problem solver outputs by $\mathbf{possible}_{(PS,I,R)}$.

We then note the following three lemmas.

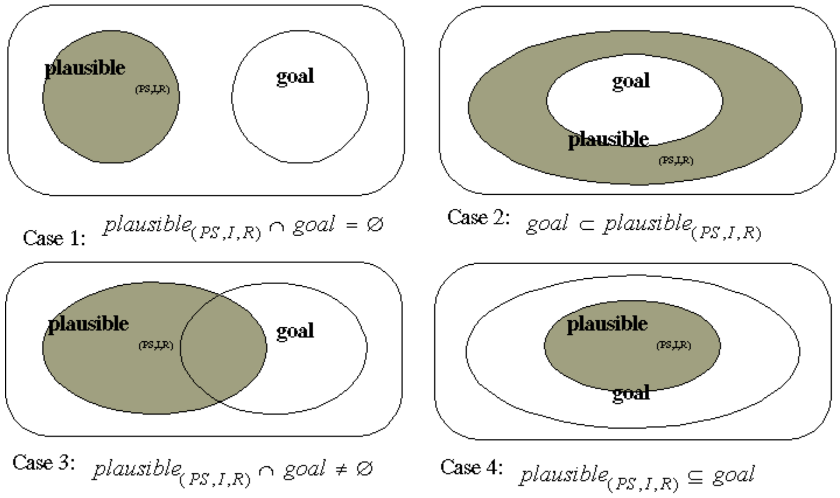

Figure 3: Inspecting the relationship between the problem solving goal and the plausibility approximation

Lemma 1: The goal is only satisfiable if

$$(\exists p)(p \in \mathbf{possible}_{(PS,I,R)}) \wedge (p \in goal)$$

or equivalently,

$$\mathbf{possible}_{(PS,I,R)} \cap goal \neq \emptyset$$

Lemma 2: If $\mathbf{possible}_{(PS,I,R)} \cap goal = \emptyset$, then the problem solver *cannot* satisfy the goal.

Lemma 3: If every possible output value satisfies the goal, i.e., $\mathbf{possible}_{(PS,I,R)} \subseteq goal$, then the goal *must* be satisfied.

The plausibility approximation to $\mathbf{possible}_{(PS,I,R)}$ is the set of values which are plausibly output by the problem solver PS, given the inputs $I = \{I_1, I_2, ...I_n\}$ in roles $R = \{R_1, R_2, ...R_n\}$. We denote this approximation by $\mathbf{plausible}_{(PS,I,R)}$, where $\mathbf{plausible}_{(PS,I,R)} \supseteq \mathbf{possible}_{(PS,I,R)}$. Let us now inspect the intersection $\mathbf{plausible}_{(PS,I,R)} \cap goal$, without additional knowledge of $\mathbf{possible}_{(PS,I,R)}$.

There are four cases to consider (see figure 3).

Case 1: Goal is *implausible* if $\mathbf{plausible}_{(PS,I,R)} \cap goal = \emptyset$. This follows because the plausible set of values contains the possible set of values (by definition), so

$$\mathbf{plausible}_{(PS,I,R)} \cap goal = \emptyset \Rightarrow \mathbf{possible}_{(PS,I,R)} \cap goal = \emptyset$$

It follows from lemma 2 that the goal is unsatisfiable.

Case 2: The goal is *plausible* if $goal \subset \mathbf{plausible}_{(PS,I,R)}$. In this case it does not necessarily follow that $goal \subset \mathbf{possible}_{(PS,I,R)}$. There are two cases to consider - either $\mathbf{possible}_{(PS,I,R)} \cap goal = \emptyset$, which would imply that the goal is unsatisfiable (from lemma 2), or $\mathbf{possible}_{(PS,I,R)} \cap goal \neq \emptyset$, which leaves the goal plausible (from lemma 1). Without knowing the contents of $\mathbf{possible}_{(PS,I,R)}$, we do not know which of the two cases applies, so we must draw the weaker of the two conclusions, i.e., that the goal is plausible.

Case 3: Goal is *plausible* if $\mathbf{plausible}_{(PS,I,R)} \cap goal \neq \emptyset$, with the same reasoning as for case 2.

Case 4: Goal is *satisfied* if $\mathbf{possible}_{(PS,I,R)} \subseteq goal$. If the plausibility approximation is completely contained within the goal, then all the possible problem solver outputs must also satisfy the goal. That is,

$$\mathbf{plausible}_{(PS,I,R)} \subseteq goal \Rightarrow \mathbf{possible}_{(PS,I,R)} \subseteq goal$$

Therefore, from lemma 3, the goal must be satisfied.

3.2 Application to Problem-Solving

We are applying these ideas to problem solving scenarios in which a number of knowledge bases $(KB_1, KB_2, ...KB_n)$ are assigned input roles [8] $(R_1, R_2, ...R_n)$ to a problem-solver, and a desired goal is stated. We call this a *problem-solver configuration* (left hand side of Figure 4). Note that at this stage we have made no assumptions about the representation of the knowledge; *when we say "knowledge base" we are not necessarily referring to a set of rules.* Nor are we making any assumptions about the *specificity* of the knowledge bases with respect to the given problem instance. That is, a knowledge base might remain constant over all problem instances, it might change occasionally, or it might change with every instance. In essence, any input to the problem-solver has been labelled as a knowledge base. We use the knowledge obtained from these inputs and a "plausibility description" of the behaviour of the problem-solver to generate a set of plausible output values. Note that under this scheme, a problem-solver "description" is itself a *program* that, given appropriate inputs, generates the problem solver's plausible output. The description therefore represents a kind of model of the problem solving task; it is a *meta* – problem-solver (right hand side of Figure 4). The purpose of building the meta – problem-solver is to determine, at low computational cost, whether the goal is consistent with the set of plausible values that it generates. If the goal is *not* consistent with this set, then it will also not be satisfied by the output of the problem-solver itself. In such cases, we need not run the problem-solver to test its outcome.

But how can this functionality be implemented? In a naive *generate-and-test* approach, we might answer the plausibility question by testing every point in the plausible output set against the goal. Unfortunately, this is both computationally inefficient (O"Hara & Shadbolt, 1996), and enables only the modelling of sets of finite size. For a more flexible and computationally more efficient version, we prefer a *constraint-based* approach [9] . The idea is that a set of plausible values is represented using constraints. The plausibility set could be of infinite, or indefinite, size, such as the set of even counting numbers, or the set of all persons whose surname begins with "W". To test the plausibility of a given goal, the goal is first represented as a set of constraints. These constraints are then applied to the plausibility set generated by a meta – problem-solver. If the resulting set is the empty set, then the goal was not plausible; if the resulting space is non-empty (or cannot easily be demonstrated to be empty), then the goal remains plausible. Furthermore, unlike the meta-level reasoning required for the halting problem, the question of plausibility, if defined appropriately, can be guaranteed to terminate within a finite number of program steps, because the meta-level need only be an *approximation* to the problem solving level [10].

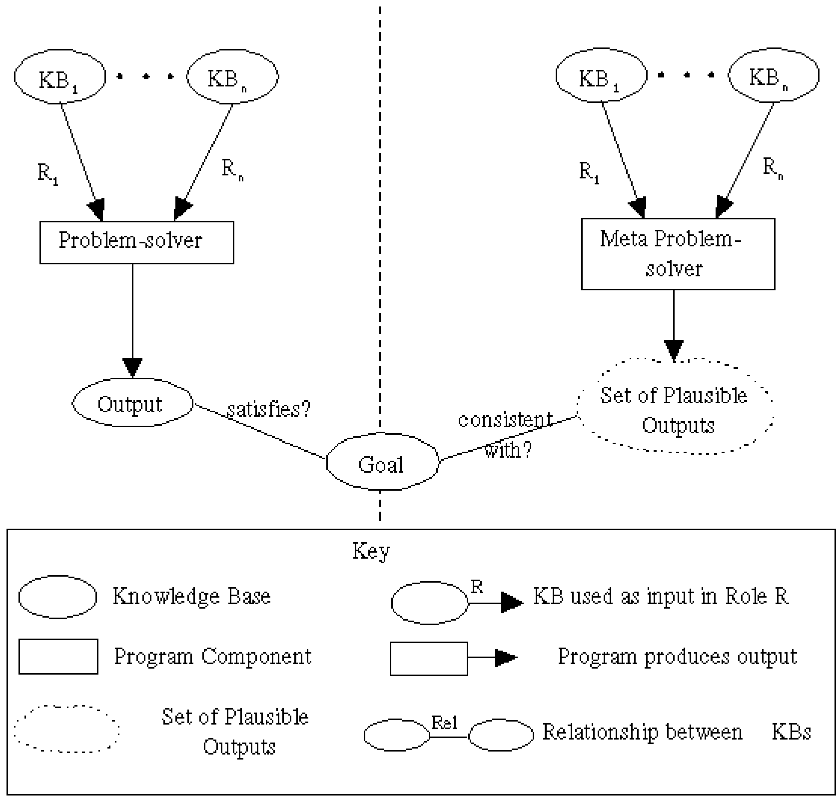

Figure 4: Problem-solver and Meta-problem-solver Configurations

Note that the plausibility refers to the combination of knowledge bases, their problem-solving roles and the specific goal. As an exploration of all the KB – role pairings might take a considerable amount of time, we make this process more efficient by assigning preconditions to each problem solving role, and first checking that these are fulfilled before conducting a plausibility test. However, note that in some cases we may still have to explore a number of KB – role combinations before deciding that no plausible solution exists.

More general scenarios are also possible, since a plausibility set generated by one meta – problem-solver can also be used as an *input* to a further meta – problem-solver. When cascading descriptions in this way, the approximate input to the second meta – problem-solver is used to generate a further approximate output. Note that the approximation generated by the second meta – problem-solver is less precise [11] than when particular inputs are given. Nevertheless, the generated plausible output set may still contain enough knowledge to answer some interesting questions about the combination of problem-solvers.

4 Implementation Approach

We mentioned earlier that we prefer a constraint-based approach to answering the question "Is the goal contained in the set of plausible output values?". In fact, it is crucial to the implementation that constraints derived from the goal can be used to reduce the size of the plausibility set *without enumerating it.* This technique makes the plausibility test less expensive than running the problem-solver because the whole search space does not have to be explored to determine whether a goal is implausible. It also leads one naturally to consider an implementation using constraint programming. As the tools we wished to model and the rest of the MUSKRAT framework were already implemented in Common LISP, we implemented the plausibility approximation to fitness-for-purpose using SCREAMER, a constraint programming library for Common LISP (Siskind & McAllester, 1993a; Siskind & McAllester, 1993b).

4.1 A Brief Introduction to SCREAMER and SCREAMER+

SCREAMER introduced two paradigms into Common LISP: firstly, it introduced a non-determinism similar to the backtracking capabilities of PROLOG. Secondly, and more importantly for our purposes, it used these backtracking capabilities as the foundation for a *declarative constraints* package. The SCREAMER constraints package provides LISP functions that enable a programmer to *create constraint variables*, (often referred to simply as *variables*), *assert constraints* on those variables, and *search for assignments of values to variables* that are consistent with the asserted constraints.

Although SCREAMER [12] forms a very good basis for solving numeric constraint problems in Common LISP , we found it less good for tackling problems based on constraints of a more symbolic nature. We therefore extended the SCREAMER library in three major directions: firstly, to improve the expressiveness of the library with respect to constraints on lists; secondly, to introduce higher order [13] constraint functions such as constraint-oriented versions of the LISP functions `every`, `some`, and `mapcar`; and thirdly, to enable constraints to be imposed on CLOS [14] objects and their slots. We called the extended library SCREAMER+. A more detailed account of SCREAMER+ is given in our technical report (White & Sleeman, 1998).

In the next sections, we outline an approach to the construction of meta – problem-solvers using SCREAMER+ , and provide an example of its usage to model the fitness-for-purpose of the set-pruning problem-solving method.

4.2 Building a Meta – Problem-Solver

To capture a description of a problem solver in terms of its inputs and outputs, we propose that the MUSKRAT developer performs the following activities.

- Elicit the **necessary properties of each of the problem solver's input roles**, and express those properties as constraints [15].
- Elicit the **properties necessary for inter-role consistency**, and express those properties as constraints across the input roles.
- Elicit **guaranteed properties of the problem solver's output(s)**, and express those properties as constraints. Often, properties of the output(s) are expressed in terms of the input roles.

These three sets of constraints then form the basis of the meta – problem-solver. The aim of the process is to collect constraints that capture detailed ("glass-box") knowledge of the problem solver. Once captured, this knowledge can subsequently be made available as "black box knowledge" to a user of the MUSKRAT toolbox. The acquisition of these constraints is a *once-only* activity because the problem solver description is also goal-independent. In other words, the same meta – problem-solver can be reused for the fitness-for-purpose verification of different knowledge bases for different tasks.

Note that initially there may be many plausible KB-role assignments, so the verification of fitness-for-purpose represents a *search* (subject to the above-given constraints) through the space of the available KBs and the known input roles of the problem solvers.

4.3 Example - Set Pruning

As an example of the approach, consider the *set pruning* method (Benjamins et al., 1999; Groot, ten Teije & van Harmelen, 1999). This method iterates over the members of a set, removing those members that do not exhibit a given property. It can therefore be regarded as a simple filter. A problem-solver with this functionality can be written in LISP as follows:

```
(defun set-prune (input-set test)
   (remove-if-not test input-set))
```

The set-pruning functionality can be demonstrated as follows:

```
;;; Create a set for testing (represented by a list)
> (setq my-set '(2 to 22 two "too"))
(2 TO 22 TWO "too")

;;; Prune away all the non-integers
> (set-prune my-set #'integerp)
(2 22)
```

The important issue here is not writing the problem solver itself, but the ability to describe the problem solver with a meta-program. We now develop a meta – problem-solver according to the guidelines set out in section 4.2. We also demonstrate how the meta – problem-solver can be used to prove the implausibility of a problem solving goal.

4.3.1 Necessary Properties of the Input Roles

The set pruner accepts two inputs: a *set* to be pruned, and a *test,* which each member of the output must satisfy. In our implementation, the set is represented by a list, and the test is represented by a function. In this case, the property for a plausible role can be succinctly represented as a LISP type – the role of a set can be defined as a synonym for the LISP type `list`:

```
(deftype plausible-set ()  'list)
```

We define a plausible test as any function or CLOS method of one argument [16] . The corresponding type for a plausible test can be defined in Allegro Common LISP as follows:

```
;;; Predicate to check whether given function accepts a single
;;; argument
(defun one-arg-functionp (f)
  (let ((args (arglist f)))
    (cond
      ((= (length args) 1) t)
      ((member (car args) '(&rest &optional)) t)
      (t nil))))

;;; Defines a specialisation of the function type for
;;; functions of one argument
(deftype plausible-test ()
  '(and function (satisfies one-arg-functionp)))
```

These definitions allow us to constrain an input to be either a plausible-set or a plausible-test.

4.4 Properties Necessary for Inter-role Consistency

For inter-role consistency, we check the mutual compatibility of the *test* and the *set.* These two inputs would be inconsistent, for example, if the *test* is for the primeness of a number [17] and the set contains strings as well as numbers:

```
(defmethod primep ((n integer))
  (setq n (abs n))
  (when (= n 2) (return-from primep t))
```

```
  (unless (evenp n)
    (do ((divisor 3 (+ divisor 2))) ;check odd numbers from 3
        ((> divisor (sqrt n)) t)   ;up to root of n
      (when (zerop (mod n divisor))
        (return-from primep nil)))))

;;; Assign test1 to be our implemented test for primeness
> (setq test1 #'primep)
#<STANDARD-GENERIC-FUNCTION PRIMEP>

;;; Assign set1 to be a plausible set which includes a string
> (setq set1 (list 0 5 "m"))
(0 5 "m")

;;; Assign test1 to be our definition of primep
> (setq test1 #'primep)
#<STANDARD-GENERIC-FUNCTION PRIME>

;;; Demonstrate that errors can occur if the test cannot be
;;; applied to every member of the set
> (set-prune set1 test1)
Error: No methods applicable for generic function
#<STAND..GEN..FN.. PRIMEP> with args ("m") of classes (STRING)
...
>
```

However, a meta – problem-solver can recognise the condition of inter-role inconsistency that causes such an error. For the example of primeness, the idea can be realised by introducing a new function, called `testable`, which returns true (T) if the given test can be applied to a supplied argument, and false (`NIL`) otherwise.

```
(defun testable (input-set test)
  ;; Conclude that test is plausible if it is
  ;; not implemented as a CLOS method
  (when (not (equal (type-of test) 'standard-generic-function))
    (return-from testable t))
  ;; Otherwise inspect the types of input that the method accepts
  ;; to determine plausibility
  (dolist (element input-set)
    (when (null (compute-applicable-methods test (list element)))
      (return-from testable nil)))
  t ; returns true by default
  )
```

The function works by considering two cases. Firstly, if the test is a "true function" (i.e., *not* a CLOS method) then LISP cannot detect the expected types of its arguments, and it is not checked further for its compatibility with the given arguments. If, however, the test is implemented as a generic function with one or more CLOS methods, then the function determines whether the generic function is applicable to each of the values in the given set [18]. That is, the function checks whether a method exists that accepts arguments of the same type as the elements of the supplied set. Note that in LISP, the generic function approach (using `defmethod`) allows for stronger typing of input arguments than the weakly typed `defun` form. The generic function approach therefore provides good support for checking inter-role consistency.

Notice that `testable` has been implemented in a very general way, and does not encode any prior knowledge of the test. Furthermore, it will also be a cheap test to execute – certainly faster than executing the test itself. (We can be sure of this because we know that the LISP system must compute the applicable methods of a generic function with any given arguments before a method can be applied. Also, as this is such an important part of LISP's run-time evaluation system, we can be confident that it has been optimised.)

Given the definition of `testable`, we can now check whether `test1` can safely be applied to each of the members of `set1`:

```
> (testable set1 test1)
NIL
```

We can also *assert* that `test1` be safely applied to all members of `set1`, as follows. Note that with the value of `set1` given above, the assertion fails, as expected:

```
> (assert! (funcallv #'testable set1 test1))
--> FAILS
```

4.4.1 Guaranteed Properties of the Problem Solver's Output(s)

When pruning a set, it is guaranteed that:

- The output is of type `plausible-set`;
- The output-set is a subset of the input-set (i.e., output-set $\subseteq$ input-set); and,
- Every member of the output-set satisfies the given test.

These three statements are captured by the following three assertions [19] :

```
(assert! (typepv output-set 'plausible-set))
(assert! (subsetpv output-set input-set))
(assert! (everyv (constraint-fn test) output-set))
```

By combining these constraints with the constraints given earlier for ensuring satisfaction of *role properties* and *inter-role consistency*, the following definition of the meta-pruner is derived:

```
(defun meta-prune (input-set test)
  (let ((output-set (make-variable)))
    ;;; Assert properties of input roles
    (assert! (typepv input-set 'plausible-set))
    (assert! (typepv test 'plausible-test))
    ;;; Assert inter-role consistency
    (assert! (funcallv #'testable input-set test))
    ;;; Assert known properties of the output
    (assert! (typepv output-set 'plausible-set))
    (assert! (subsetpv output-set input-set))
```

```
    (assert! (everyv (constraint-fn test) output-set))
    output-set) ; output-set is the return value
  )
```

In this definition, the inputs `input-set` and `test` correspond to the inputs KB_1 and KB_2 of figure 4. The meta–problem-solver first generates a constraint variable to be used as its output, then asserts the necessary properties of its input roles. Next, it guarantees the inter-role consistency of its inputs by asserting that the supplied test must be applicable to each member of the input-set. Finally, the properties of the output are asserted; namely, that each member of the output-set is also a member of the input-set, and that the supplied test is satisfied by each member of the output-set. Note also that a meta – problem-solver must return the constraint variable which represents the plausible output value of the problem solver.

4.4.2 Verifying Fitness-for-Purpose

The meta – problem-solver is interesting because it enables reasoning about the output of the problem solver *without executing it.* In particular, many cases in which the problem solver would not satisfy the goal are detected early; for example even before the inputs to the problem solver are considered. We now describe some examples in which failures are detected by the meta-pruner.

In the following example, the test `#'integerp` *can* meaningfully be applied to all the members of the input set, but the goal data structure '(3) is rejected because it is not a subset of the given input (recall that `set1` has the value `'(0 5 "m")`).

```
;;; The return value of the meta-pruner is a constraint
;;; variable of the type plausible-set. Every constraint
;;; variable is also assigned a unique identifier,
;;; in this case, 20.

> (setq plausible-output (meta-prune set1 #'integerp))
[20 plausible-set]

> (assert! (equalv plausible-output '(3)))
--> FAILS
```

In the next example, the symbolic argument `'not-a-function`, representing the `test`, is not a function, and is therefore incompatible with the expected type `plausible-test`:

```
> (meta-prune '(0 5 "m") 'not-a-function)
--> FAILS
```

In the following example, the supplied test (for whether an integer is prime) cannot meaningfully be applied to all the members of the input-set:

```
> (meta-prune '(0 5 "m") #'primep)
--> FAILS
```

4.4.3 Searching for Plausible Configurations

Suppose that several input-sets were available, as well as a single goal. By using SCREAMER+ , together with the meta-pruner, it is possible to discover which of the input-sets plausibly satisfy the goal. For example, suppose that the input-sets were {}, {0, 1, 2}, {A, B, C}, {0 A}; and that the test prunes away items that are not integers. Suppose further that the goal is to obtain the set {0}.

Then the problem is first set up as follows:

```
;;; Set up a constraint variable that is one of the given sets

> (setq kbs (a-member-ofv '(() (0 1 2) (a b c) (0 a))))
[69 nonnumber enumerated-domain:(NIL (0 1 2) (A B C) (0 A))]

> (setq goal '(0))
(0)

> (assert! (equalv (meta-prune kbs #'integerp) goal))
NIL
```

Notice that no failures have been generated so far, indicating that the task instance is still plausible. The following SCREAMER search idiom returns all input-sets that *plausibly* satisfy the goal:

```
> (all-values (solution kbs (static-ordering #'linear-force)))
((0 1 2) (0 A))
```

In effect, the MUSKRAT method is recommending to try the input-sets {0 1 2} and {0 A} to satisfy the goal. It has filtered out the empty set {} and the set {a b c} because they do not plausibly satisfy the goal. Notice that {0 1 2} does not actually satisfy the goal, but {0 A} does:

```
;;; returns nil because
;;; (set-prune '(0 1 2) #'integerp) returns '(0 1 2)

> (equal (set-prune '(0 1 2) \#'integerp) goal)
NIL

;;; returns t because (set-prune '(0 A) #'integerp)
;;; returns '(0), which is the same as the goal

> (equal (set-prune '(0 A) #'integerp) goal)
T
```

4.5 Example - Meal Planning

Suppose we are applying a problem-solver which constructs solutions consistent with some given domain and temporal constraints (a design task) [20]. We have chosen to apply the problem-solver to the domain of meal preparation because the domain is rich and challenging, but also easily understood. Furthermore, the problem-solving task is analogous to flexible design and manufacturing, also called *just-in-time manufacturing.* In the domain of meal preparation, the *designer* is used to compose (or "construct") a dish

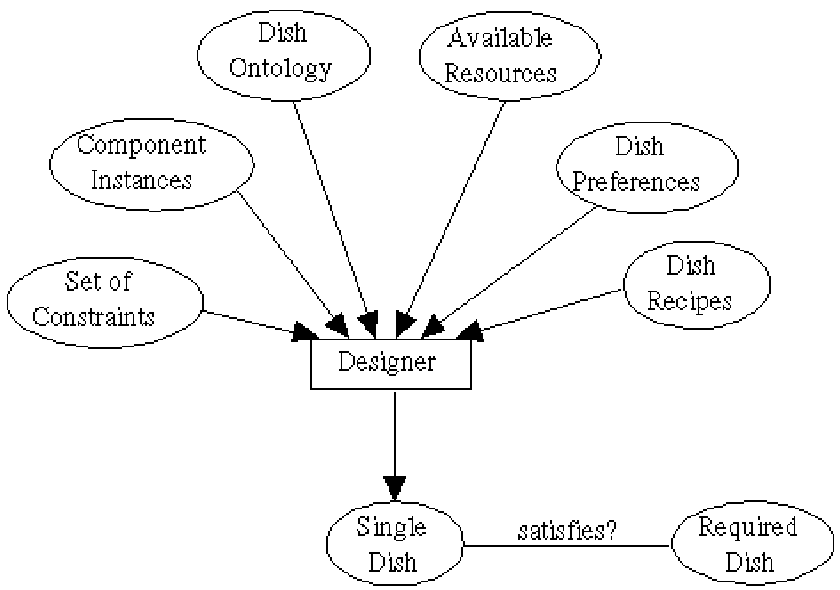

Figure 5: Problem Solving in the Domain of Meal Preparation

which is consistent with culinary, temporal, and resource constraints, also taking a set of dish preferences into consideration (see Figure 5).

Now suppose that we would like to use plausibility modelling on this problem-solver by constructing its corresponding meta – problem-solver. A *meta-designer* is easy to construct if we neglect the constraints, dish preferences, and the available resources. (Note that neglecting knowledge of inputs will have the effect of producing more general outputs – behaviour that is consistent with the approximation described in section 3.) The meta-designer produces a set of plausible dishes, together with their plausible preparation times. These times are represented as upper and lower bounds on the preparation of whole dishes, and are derived from knowledge of the preparation times of dish components. The upper bound of the dish preparation time is set as the sum of the durations of the composite tasks; the lower bound is the duration of the lengthiest task. Constraints from the goal can then be applied to the set of plausible scheduled dishes to see if inconsistencies can be detected (see Figure 6). Note that although we advocate the use of the constraint satisfaction paradigm for the implementation of the meta-level, this choice is independent of the implementation method used by the problem solver itself. The independence arises because the meta-layer treats the problem solver as a black box, describing only its input/output relationship and not how it works internally.

We have implemented the example using the knowledge provided in Table 1: a list of dish components, together with their component type, a measure of their dietary value, their cost and preparation time. In addition, the designer must be told what it should construct from this knowledge. This information is provided by a dish ontology which states that a dish consists of a main component, together with a carbohydrate component and two other vegetables. Likewise, the cost of a dish is defined to be the sum of the costs of its component parts, and the number of diet points associated with a dish is the sum of the diet points of its components. The meta-designer

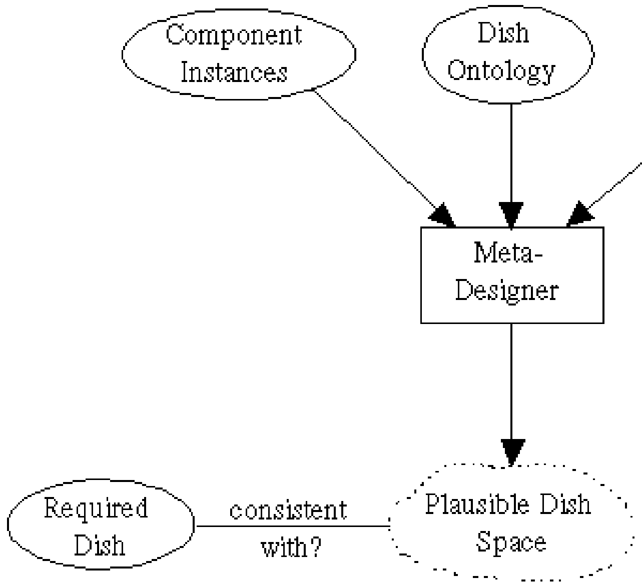

Figure 6: Plausibility Reasoning in the Domain of Meal Preparation

generates upper and lower bounds for the preparation time of a dish, as described in the previous section.

The SCREAMER+ implementation makes use of the CLOS facilities offered by defining the set of plausible dishes as a *single object* in which each *slot* (such as the main component of the dish, the preparation time, dietary points value, and cost) is constrained to be plausible. In practice, this means that constraints are expressed across the slots within a single object. If the domain of one of the slot's constraint variables changes, then this causes propagation to the other related slots.

For example, let us inspect some plausible dish preparation times and costs:

```
;;; Create an instance of the object
;;; An after method asserts the appropriate constraints across
;;; the object's slots at object creation time.

USER: (setq my-dish (make-instance 'plausible-dish))
#<PLAUSIBLE-DISH @ #x99c86a>

;;; Retrieve the value of the duration slot
USER: (slot-value my-dish 'duration)
[774 integer 10:120]

;;; Retrieve the value of the cost slot
USER: (slot-value my-dish 'cost)
[769 integer 5:7 enumerated-domain:(5 6 7)]
```

Ingredient name	**Ingredient type**	**Diet Points**[a]	**Cost (£)**	**Prep'n Time**
Lamb Chop	Main (meat)	7	4	20
Gammon Steak	Main (meat)	10	4	15
Fillet of Plaice	Main (fish)	8	4	15
Sausages	Main (meat)	8	2	10
Chicken Kiev	Main (meat)	8	3	30
Quorn Taco	Main (vegetarian)	6	3	10
Jacket Potato	Carbohydrate	5	1	60
French Fries	Carbohydrate	6	1	15
Rice	Carbohydrate	6	1	15
Pasta	Carbohydrate	2	1	10
Runner Beans	Vegetable	1	1	10
Carrots	Vegetable	1	1	15
Cauliflower	Vegetable	1	1	10
Peas	Vegetable	2	1	3
Sweetcorn	Vegetable	2	1	5

Table 1. Meal Preparation Knowledge Used in Plausibility Set Generation

a. Points schemes like these are commonly used by slimmers as a simplification of calorific value. When using such schemes, slimmers allow themselves to consume no more than, say, 40 points worth of food in a single day (the actual limit usually depends on the slimmer's sex, height, and weight).

This tells us that a plausible dish takes between ten minutes and two hours to prepare, and costs £5, £6, or £7. Now let us restrict our search to a fish dish, and again inspect the plausible values:

```
USER: (assert! (typepv (slot-valuev my-dish 'main) 'fish))
NIL

USER: (slot-value my-dish 'duration)
[774 integer 15:105]

USER: (slot-value my-dish 'cost)
7
```

The plausible range size for the preparation time of the dish has decreased, and the cost of the dish has become bound without having to make any choices of vegetables. If the goal were to have such a fish dish ready in less than quarter of an hour, it would fail without having to know the details of the dish preferences or kitchen resources used by the actual problem-solver, and without needing to run the problem-solver:

```
USER: (assert! (<v (slot-valuev my-dish 'duration) 15))
 --> FAILS
```

In a similar way, if we define a quick dish to be one that takes less than half an hour to prepare, and a "healthy" dish to be one that has a dietary points count of 16 or less, then it is easy to discover that there is no such thing as a quick dish that includes a jacket potato, or a "healthy" dish that includes gammon and french fries!

5 Discussion

In this section we discuss the contributions which MUSKRAT is making and indicate some possible directions for further work.

5.1 Contributions

MUSKRAT's contributions include the following:

1. A notation for representing the capabilities of problem-solvers, together with their relationships to the contents of knowledge sources and a desired goal;

2. The further development of the technology of constraint satisfaction through our extension to the SCREAMER package;

3. Formulation of the idea of "economical" problem solving and the investigation of its relationship to the description of problem solving methods;

4. Progress towards the provision of a framework that unifies problem solving, knowledge acquisition and knowledge transformation/refinement;

5. The development of a method to determine whether it is plausible/likely that knowledge bases can be reused by a particular problem solver to solve a particular task.

Our notation for representing the competencies of problem solvers, and the technology we have chosen for proving their properties differs considerably from the related works of Wielinga et al.(Wielinga, Akkermans & Schreiber, 1998), Fensel et al.(Fensel & Schönegge, 1997; Fensel & Schönegge, 1998), and Pierret-Golbreich (Pierret-Golbreich, 1998). The differences have evolved because we have been working towards a different, but associated, objective. Unlike the methodology of Wielinga et al., for example, we assume our problem solvers exist and seek to describe them as they are, rather than attempting to construct them as we would like them to be. Furthermore, since we are building an advisory system for novice users of problem solvers, we have strong requirements for *operational* descriptions, and a fully *automatic* proof technique. Pierret-Golbreich argues that formal methods should be used to describe problem-solving methods and that this offers a means to decide on a component's reusability. Since such methods are not operational, however, this approach cannot be applied to our problem. Fensel & Schönegge have operationalised their proof technique, but its powerful proof mechanisms are necessarily interactive (Fensel & Schönegge, 1997), and not suitable to be driven by novices. Our notation, on the other hand, is declarative, operational, and sufficiently expressive to enable the automatic derivation of interesting properties of a problem solver and the knowledge it employs.

Our extension to the SCREAMER package, driven by our requirement for a declarative and operational description of problem solvers and knowledge sources, has reached a level of expressiveness that contributes to the field of constraint technology. SCREAMER+ can be used to provide elegant solutions to many of the non-trivial problems cited as soluble by commercially available systems such as CHIP (Simonis, 1995) and ECLIPSE (Wallace, Novello & Schimpf, 1997). The LISP-based nature of SCREAMER+, however, has also helped it to gain advantages over its PROLOG-based counterparts, such as the ability to assert constraints on expressions that take *functions* as arguments. This is important in the current context because a function might be the realisation of a PSM.

Our ideas of economical problem solving have been motivated by the requirement to generate advice on the suitability of problem solvers "on the fly" at run-time. Conceptually, some of the techniques applied can be seen variously as applications of abstraction (Giunchiglia & Walsh, 1992), or special cases of generalisation transformations within the plausibility framework of Collins and Michalski (Collins & Michalski, 1989; Oroumchian, 1995). In contrast to these approaches, which classify their methods according to the morphology of their mappings, we are investigating the knowledge-level dependencies of each technique in terms of domain, task, and method.

The fourth contribution of MUSKRAT is to create a unified framework for problem solving, directed knowledge acquisition, and knowledge transformation. This aim is shared by the generalised directive models (GDM) work of O"Hara, Shadbolt and van Heijst (O"Hara, Shadbolt & van Heijst, 1998), but there are significant differences as their work takes place in the context of analysing source materials and attempting to co-evolve domain ontologies as well as the appropriate model for the task; in some cases, a KA task is activated. In the case of MUSKRAT , the task to be solved and the problem solver to be used have already been identified. MUSKRAT seeks to discover whether existing knowledge bases can be used without change with the identified problem solver, whether it can be transformed before use, or whether it is necessary to acquire a completely new knowledge base using a KA tool. Other work that has sought to provide a common framework for problem solving and KA includes MOBAL (Morik et al., 1993), VITAL (Motta, O"Hara & Shadbolt, 1996), and NOOS (Arcos & Plaza, 1994; Arcos & Plaza, 1997).

Our contribution to the topic of knowledge reuse is most closely related to Puppe's work (Puppe, 1998) and the Protégé project (Gennari et al., 1998). Puppe has realised, as do O'Hara et al.(O'Hara, Shadbolt & van Heijst, 1998) that as experts learn more about the domain, their perspectives on the task may change, and hence it is highly desirable that acquired knowledge can be used with a different algorithm from the one for which it was initially acquired. Puppe confines his attention to classification tasks, and notes a number of different classification algorithms that process the same information in different ways. He discusses a number of algorithms including CATEGORICAL (which produces decision trees/tables), heuristic classifiers (which use heuristic rules) and case-base reasoners. Puppe's important insight was to reimplement several classifiers so that they have common data files that can be readily reused. In MUSKRAT , however, we have set up a more general framework in which a knowledge base initially used with a PSM for a classification task might be reused with a different PSM for synthesis or planning.

Protégé's long-term goal is to build a tool-set and methodology for the construction of domain-specific KA tools and knowledge-based systems from reusable components. The project plans to develop a variety of methods and knowledge bases, all packaged as CORBA (Common Object Request Broker Architecture) components. Knowledge bases are made available on a server which uses the OKBC (Open Knowledge Base Connectivity) protocol, enabling developers to query the frame-based knowledge with functions such as `get-class-all-subs` to retrieve classes, `get-frame-slots` to retrieve slots, and `get-class-all-instances` to retrieve the instances of a class. A single method, `propose-and-revise`, is available within the framework to date, but there are several knowledge bases which have been used for the VT (elevator-configuration) task, U-Haul (truck selection), as well as the Ribosome and tRNA configuration tasks. In order for a particular method to reuse knowledge from a particular knowledge base a mediator must be manually encoded for each method/KB pair. However, Gennari et al. (Gennari et al., 1998) argue that given the similarities in the representations of the knowledge bases, there will be many commonalities between the various mediators needed. This already reduces the workload required to produce the mediators, and the Stanford group envisages partially automating mediator production in the future. MUSKRAT's third case (see section 1) corresponds to the situation for which Protégé currently creates mediators. Whilst Protégé *assumes* that a mediator is necessary, MUSKRAT has evolved a test to determine whether an available knowledge base might be *directly* reusable for the given task. If this is the case, no mediation or adaptation is required.

5.2 Further Work

This paper has described our approach to fitness-for-purpose and shown how it can be applied to relatively small-scale examples. We also plan to use the same approach on larger, more realistic problems.

The arithmetic example of fitness-for-purpose presented in section 2 showed how properties of parity in integer arithmetic can be used to assess the plausibility of an arithmetic equation. Arithmetic expressions were abstracted to their equivalent parity expressions before the precondition was applied. Moreover, there was a very close relationship between the precondition and the original arithmetic expression. In such cases, it is possible for the precondition to be generated automatically (Robertson, 1999). This represents a class of tasks to be investigated.

As an example, suppose again that we would like to compute the plausibility of the arithmetic expression $22 \times 31 + 11 \times 17 + 13 \times 19 = 1097$. That is, the left-hand side forms the input to the meta – problem-solver, and the *goal* is that the expression plausibly evaluates to the right-hand side. We can first rewrite the expression into LISP's prefix notation, giving

```
(= (+ (* 22 31) (* 11 17) (* 13 19)) 1097)
```

Then, since the meta-level in this case applies a different algebra to the same numbers, the corresponding meta-level expression can be generated ("lifted") by replacing the ground-level operators (+, * and =) by their meta-level counterparts (`meta+`, `meta*` and `meta=`), giving

```
(meta= (meta+ (meta* 22 31) (meta* 11 17) (meta* 13 19)) 1097)
```

With appropriate constraint-based definitions of the meta-level functions (given in White, 2000), the expression fails, signifying that the goal is inconsistent with the value of the arithmetic expression:

```
> (assert! (meta=
       (meta+ (meta* 22 31) (meta* 11 17) (meta* 13 19))
       1097))
--> FAILS
```

The call to `meta=` represents the consistency check between the problem solver's output and the goal. Note that the precondition to apply to arithmetic expressions varies with the task instance, so it is imperative that the precondition be automatically computed as part of the meta – problem-solver.

The approach to fitness-for-purpose described in this paper is a binary decision process. That is, a configuration of problem solver and knowledge bases is deemed to be either (plausibly) fit for solving a goal, or unfit. This is somewhat unsatisfactory, because it provides no information about how much work would need to be done to transform an unfit configuration into a fit one. Rather than introduce a range of numbers that indicate the level of fitness, but do not state where the problems lie, we propose to extend the current model by introducing a lattice (i.e., a hierarchy with multiple inheritance) of problem solving roles. As we have seen, problem solving roles constrain the knowledge they contain to be fit for consumption by a problem solver. The extension would mean organising the criteria, represented by constraint objects, into a multiple inheritance hierarchy. The objects near the top of the hierarchy are weakly constrained, whereas those nearer the leaves are strongly constrained. A fitness-for-purpose search would traverse this hierarchy, looking for the strongest set of criteria which are fulfilled by a knowledge base. That is, if a knowledge base fails to satisfy a given role, the search could move to a weaker role (parent object), to see if it is satisfied by the knowledge base. If this role succeeds, the approach would be able to identify those criteria which are satisfied by the knowledge base, and those which are not. This provides significant information to a user who faces the prospect of fixing a knowledge base to solve the problem.

5.3 Summary

In this paper, we defined the *fitness-for-purpose* of a problem-solving configuration, which consists of a problem-solver, knowledge bases, and a goal. We argued that the *plausibility* of a configuration is a tractable approximation to fitness-for-purpose. Constraint logic programming offers a promising approach to the implementation of plausibility tests because it enables a knowledge engineer to write a declarative *meta* – problem-solver, which makes statements about the relationship between the inputs and outputs of the actual problem-solver. When this knowledge is coupled with the knowledge of some goal that the user is trying to achieve, it reduces the space of plausible results in a way that can lead to early failures. A failure denotes the *implausibility* of the task, a property that can be detected without recourse to running the problem solver.

We believe the notion of fitness-for-purpose to be important for knowledge acquisition and other fields of software engineering. However, we recognised that such a notion would be of little use without an operational method to support its detection. Our investigation of possible approaches and the construction of the method raised many interesting research questions. Moreover, our divide-and-conquer approach to the requirements of MUSKRAT (see cases 1, 2 & 3 in section 1) provides a basic "road map" for the further exploration of related issues.

5.4 Acknowledgements

This work was financially supported by an EPSRC studentship. The MUSKRAT framework was initially conceived by Nicolas Graner; further inspiration came from the Machine Learning Toolbox Project (ESPRIT project 2154). We also gratefully acknowledge the comments of Frank van Harmelen and Ken Brown.

References

1. Arcos, J. L., Plaza, E., (1994), "Integration of Learning into a Knowledge Modelling Framework", in Proceedings of the Eighth European Knowledge Acquisition Workshop (EKAW "94), LNCS, Springer Verlag.

2. Arcos, J. L., Plaza, E., (1997), "Noos: An Integrated Framework for Problem Solving and Learning", Research Report 97-02, Institut d'Investigació en Intelligència Artificial (IIIA), Barcelona, Spain.

3. Benjamins, V. R., Fensel, D., (eds.), (1998), International Journal of Human-Computer Studies (1998), Special Issue on Problem Solving Methods, Vol. 49, No. 4.

4. Benjamins, V. R., Plaza, E., Motta, E., Fensel, D., Studer, R., Wielinga, B., Schreiber, G., Zdrahal, Z., Decker, S., (1998), "IBROW3 – An Intelligent Brokering Service for Knowledge Component Reuse on the World-Wide Web", in proceedings of the Eleventh Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW98), Banff, Alberta, Canada.

5. Benjamins, V. R., Wielinga, B., Wielemaker, J., Fensel, D., (1999), "Brokering Problem Solving Knowledge on the Internet", in the proceedings of the Eleventh European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW "99), LNCS, Springer Verlag.

6. Brassard, G., Bratley, P., (1996), "Fundamentals of Algorithmics", Prentice-Hall International, pp. 160-161.

7.Collins, A., Michalski, R. S., (1989), "The Logic of Plausible Reasoning: A Core Theory", Cognitive Science, Vol. 13, pp. 1-49.

8. Fensel, D., Schönegge, A., (1997), "Using KIV to Specify and Verify Architectures of Knowledge-Based Systems", in Proceedings of the Twelfth International Conference on Automated Software Engineering (ASEC-97), Incline Village, Nevada.

9. Fensel, D., Schönegge, A., (1998), "Inverse Verification of Problem Solving Methods", International Journal of Human-Computer Studies, Vol. 49, No. 4, pp. 339-361.

10. Gennari, J. H., Cheng, H., Altman, R. B., Musen, M. A., (1998), "Reuse, CORBA, and Knowledge-based Systems", International Journal of Human-Computer Studies, Vol. 49, No. 4, pp. 523-546.

11. Giunchiglia, F., Walsh, T., (1992), "A Theory of Abstraction", Artificial Intelligence, Vol. 56, No. 2-3, pp. 323-390.

12. Graner, N., Sleeman, D., (1993), "MUSKRAT: A Multistrategy Knowledge Refinement and Acquisition Toolbox", in proceedings of the Second

International Workshop on Multistrategy Learning, R. S. Michalski and G. Tecuci (Eds.), pp. 107-119.

13. Hameed, A., Sleeman, D., & Preece, A. (2001), "Detecting Mismatches Among Experts" Ontologies Acquired through Knowledge Elicitation", to appear *in Research and Development in Intelligent Systems XVIII (Proceedings of ES2001)*: the 21st SGES International Conference on Knowledge Based Systems and Applied Artificial Intelligence, Cambridge, U.K. BCS Conference Series, Springer Verlag.

14. Imielinski, T., (1987), "Domain Abstraction and Limited Reasoning", in Proceedings of the Tenth International Joint Conference on Artificial Intelligence, pp. 997-1003.

15. Johnson, J., (1997), "Mathematics, Representation, and Problem Solving", Mathematics Today (Bulletin of the Institute of Mathematics and its Applications), Vol. 33, No. 3., pp. 78-80.

16. O'Hara, K., Shadbolt, N., (1996), "The Thin End of the Wedge: Efficiency and the Generalised Directive Model Methodology", in Shadbolt, N., O"Hara, K., Schreiber, G., (Eds), Advances in Knowledge Acquisition, proceedings of the 9th European Knowledge Acquisition Workshop (EKAW "96), Nottingham, UK, pp. 33-47.

17. O'Hara, K., Shadbolt, N., van Heijst, (1998), "Generalised Directive Models: Integrating Model Development and Knowledge Acquisition", International Journal of Human-Computer Studies, Vol. 49, No. 4, pp. 497-522.

18. Morik, K., Wrobel, S., Kietz J-U., Emde, W., (1993), "Knowledge Acquisition and Machine Learning: Theory, Methods and Applications", Academic Press, London.

19. Motta, E., O"Hara, K., Shadbolt, N., (1996), "Solving VT in VITAL: A Study in Model Construction and Knowledge Reuse", International Journal of Human-Computer Studies, Vol. 44, No. 3, pp. 333-371.

20. Oroumchian, F., (1995), "Theory of Plausible Reasoning", in Information Retrieval by Plausible Inferences: An Application of the Theory of Plausible Reasoning of Collins and Michalski, PhD Thesis, School of Computer and Information Science, Syracuse University, New York.

21. Pierret-Golbreich, C., (1998), "Supporting Organization and Use of Problem-solving Methods Libraries by a Formal Approach", International Journal of Human-Computer Studies, Vol. 49, No. 4, pp. 471-495.

22. Polya, G., (1957), "How To Solve It: A New Aspect of Mathematical Method", Doubleday Anchor Books, New York.

23. Puppe, F., (1998), "Knowledge Reuse among Diagnostic Problem-Solving Methods in the Shell-Kit D3", International Journal of Human-Computer Studies, Academic Press, Vol. 49, No. 4, pp. 627-649.

24. Robertson, D., (1999), Personal Communication.

25. Schreiber, G., Akkermans, H., Anjewierden, A., De Hoog, R., Shadbolt, N., Van de Velde, W., Wielinga, B., (1999), "Knowledge Engineering and Management: The CommonKADS Methodology", MIT Press, Cambridge, MA, USA.

26. Simonis, H., (1995), "The CHIP System and Its Applications", in Montanari, U., Rossi, F., (Eds.), Principles and Practice of Constraint Programming, proceedings of the First International Conference on the Principles and Practice of Constraint Programming, Lecture Notes in Computer Science Series, Springer Verlag, pp. 643-646.

27. J. M. Siskind, D. A. McAllester, (1993a), "SCREAMER: A Portable Efficient Implementation of Nondeterministic Common LISP"', Technical Report IRCS-93-03, University of Pennsylvania Institute for Research in Cognitive Science.

28. J. M. Siskind, D. A. McAllester, (1993b), "Nondeterministic LISP as a Substrate for Constraint Logic Programming", in proceedings of AAAI-93.

29. Sleeman, D., White, S., (1997), "A Toolbox for Goal-driven Knowledge Acquisition", in proceedings of the Nineteenth Annual Conference of the Cognitive Science Society, (COGSCI '97), Stanford, CA.

30. Turing, A. M., (1937), "On Computable Numbers, with an Application to the Entscheidungsproblem", in Proceedings of the London Mathematical Society, Vol. 42(ii), pp. 230-265; correction Vol. 43, pp. 544-546.

31. Wallace M. G., Novello, S. and Schimpf, J., (1997) "ECLIPSE: A Platform for Constraint Logic Programming", ICL Systems Journal, Vol. 12, Issue 1, May 1997.

32. White, S., Sleeman, D., (1998), "Providing Advice on the Acquisition and Reuse of Knowledge Bases in Problem Solving", in proceedings of the Eleventh Banff Knowledge Acquisition for Knowledge-Based Systems Workshop (KAW98), Banff, Alberta, Canada.

33. White, S., Sleeman, D., (1998), "Constraint Handling in Common LISP", Technical Report AUCS/TR9805, Department of Computing Science, University of Aberdeen, Scotland, UK.

34. White, S., (2000), "Enhancing Knowledge Acquisition with Constraint Technology", PhD Thesis, Department of Computing Science, University of Aberdeen, Scotland, UK.

35. White, S., Sleeman, D., (2001), "A Grammar-driven Knowledge Acquisition Tool that incorporates Constraint Propagation", to appear in proceedings of K-CAP 2001: The First International Conference on Knowledge Capture, Victoria, British Columbia, Canada.

36. Wielinga, B. J., Akkermans, J. M., Schreiber A. Th., (1998), "A Competence Theory Approach to Problem Solving Method Construction", International Journal of Human-Computer Studies, Vol. 49, No. 4.

Footnotes

[1] Dr. Simon White is also affiliated with the Computing Science Department, University of Aberdeen, Scotland, UK.

[2] For simplicity, we currently assume the application of a single, chosen problem-solver.

[3] The truth-table for logical implication ($\rightarrow$) defines both `false` $\rightarrow$ `true` and `false` $\rightarrow$ `false` as `true`.

[4] Many AI programs perform searches of problem spaces that grow exponentially with the size of the problem.

[5] One way to determine whether the plausibility test is useful is to compare the computational complexities of the problem solver and the plausibility test. If the complexity of the plausibility test is lower than that of the problem solver, we might assume it is reasonable to apply the plausibility test first. Unfortunately, this model takes no account of the utility of the information gained from the plausibility test.

[6] Subsequent reasoning may later reveal it to be implausible.

[7] We use this notation because the output set is dependent on the problem-solver, its inputs and their roles.

[8] A role represents the way in which an input is used for problem solving. Our notion of a role corresponds very closely to the notion of a knowledge role in CommonKADS (Schreiber et al., 1999, p. 105).

[9] Consider the whimsical analogy of a man wishing to buy a pair of shoes: he does not walk into a book shop, pick up every book, and inspect it for its shoe-like qualities (the generate-and-test approach). Instead, he recognises from the outset that bookshops sell books (i.e. the plausibility set is a set of books), and that this is not consistent with his goal of buying shoes.

[10] Consider the case of the halting problem, in which we should like to determine whether a program runs forever or halts. At the meta-level, we may choose to run the program for some given length of time to see what happens. If the program halts within the allotted time, then we know that it halts. Otherwise it remains plausible that it runs forever.

[11] strictly speaking, this should read "no more precise".

[12] Version 3.20; available at the time of writing from
`http://www.cis.upenn.edu/˜screamer-tools/home.html`

[13] A "higher order function" is a function that accepts another function as an argument.

[14] CLOS is the *Common LISP Object System*, a specification of functions to enable object-oriented programming in LISP, which was adopted in 1988 as part of the Common LISP standard.

[15] The constraints represent the preconditions on roles first mentioned on page PAGEREF rolePreconditions 9.

[16] Note that technically, CLOS methods are also classified as functions.

[17] We make the distinction between the property of *primeness*, which can only meaningfully be applied to an integer; and the property of being a *prime number* , which can be applied to any type (but non-integers would always return false). As the given test is for the primeness of an integer, inconsistencies can arise and a test for inter-role consistency is appropriate.

[18] We also considered an alternative implementation, which only computed the applicability of the test for each *distinct element type* in the given set. This was a promising approach because an input set may contain many elements of the same type, and the applicable methods would be computed once for each type instead of once per member of the set. However, our implementation was no faster than the given implementation – probably

because determining the type of a LISP value is just as computationally expensive as computing a generic function's applicable methods.

[19] constraint-fn accepts a conventional LISP function as an argument, and returns a constraint-based function as its result.

[20] The task is also soluble by two separate problem solvers: the domain constraints are solved by the first, and the temporal constraints satisfied by the second. A detailed discussion of the cascading of problem solvers is outside the scope of this paper.

[21] Points schemes like these are commonly used by slimmers as a simplification of calorific value. When using such schemes, slimmers allow themselves to consume no more than, say, 40 points worth of food in a single day (the actual limit usually depends on the slimmer's sex, height, and weight).