Combining Machine Learning and Theorem Proving in Reformulating Constraint Satisfaction Problems

John Charnley
September 2005
Department of Computer Science,
Imperial College of Science,
Technology and Medicine
University of London
180 Queen's Gate
London
SW7 2AZ
jwc04@doc.ic.ac.uk

Supervisor
Dr Simon Colton
sgc@doc.ic.ac.uk

Submitted in partial fulfilment of the requirements for the MSc Degree in Computing Science of the University of London and for the Diploma of Imperial College of Science, Technology and Medicine.

Acknowledgements

I would like to thank Dr Simon Colton and Dr Ian Miguel for their help during this project. My thanks also to Alison for her support and patience and to Emily for constantly reminding me that, while things at times may seem difficult, they're worth it in the end.

Abstract

Constraint satisfaction problems are an important class of problems with many different applications. Reformulating such problems can lead to improvements in solving efficiency and allow larger and more difficult problems to be attempted. Their importance and prevalence in so many areas means that such improvements in efficiency are highly valuable. A number of different approaches to improving efficiency of have been investigated and they are a very active area of research within Artificial Intelligence.

One particular method of improving the efficiency of a constraint satisfaction problem solver is to discover additional facts about the domain of interest that could be given to the solving agent. By carefully selecting additional information, and formulating in a way that the solver can understand, the solver's task can be made easier. One method of obtaining that additional information would be to use machine learning techniques to investigate the domain and search for this additional information. Such an approach was developed in [2]. In this approach, some of the tasks were performed manually and some of the tasks were performed automatically by computer systems. For instance, the machine learning of new information was performed by the HR machine learning system but its analysis and conversion into constraints for the solver system was a manual process. The method was very successful and showed that it was possible to decrease the time solvers took to find solutions by adding correctly formulated new information. However, the approach was only semi-automatic and we were able to identify a number of areas where it could be improved. We designed processes to fully automate all those aspects of the system that were previously manual. We believed full automation would save time and make the approach less reliant on specialist knowledge.

The HR system produces output in the form of first order logic theorems. We developed a Definite Clause Grammar and other bespoke routines that are capable of translating such output into constraints understood by the solver system. Secondly, we designed a method to assess the impact of this new information upon the performance of a constraint solver. This process performs multiple reformulations of a starting solver formulation and compares the performance of each to find the best. We also developed automation for configuring the HR system for a discovery run.

We built a Java application that allows all aspects of this process to be performed either interactively or in batch mode, which encompassed developing an interface with a prolog system. We used this system to perform investigations in areas of finite algebra. We started with a naive solver formulation for each algebra and, in the majority of these experiments we demonstrated how our system could improve this naive first attempt. In addition, we identified some interesting mathematical properties of these algebras. We therefore proved that the fully automated method is sound in principle and we suggest further areas where the method can be improved and directions for its future development.

Contents

Chapter 1

Introduction

1.1 Motivation

Constraint Satisfaction Problems (CSPs) is the generic name given to problems characterised by the need to assign values to a set of variables subject to some restrictions. CSPs appear in a large number of important areas. For example, CSPs appear in scheduling, where a number of tasks have to be performed subject to some limitations. An example of this is in a factory, where resource decisions affect profitability and a CSP problem would be how best to allocate resources, for instance machine time, to maximise overall factory profitability. Another application would be in airport scheduling, where the aim would be to balance passenger safety and other restrictions, such as the time at which flights may arrive or depart, with maximising throughput at the airport. More details of CSPs are provided in §2.1 and the tools used to solve CSPs are discussed in §2.2.

Formulating CSPs correctly is a difficult and skilled task requiring a great deal of expertise and time. Great value is placed, therefore, on any methods which can assist in reducing the time or expertise required by this process. In addition, once formulated, a CSP problem can take a large amount of computer processing time to solve as there may be a huge number of combinations to be considered by the computer in searching for a solution. Such computing time is costly and therefore a great deal of research has been performed into different methods of reducing this processing time.

One method of improving the effectiveness of a solver agent and reducing the time required to find solutions is to provide new information, in the form of additional constraints, to the solver process. Such information may then reduce the search space that the solver needs to explore or better guide its search for solutions. In [2], Colton and Miguel used this approach to perform reformulations of CSP problems, see §2.6. They used the HR machine learning system, described in §2.5, to discover new information about the problem domains based on a limited starting knowledge of the domain. They reviewed the information that HR produced to identify anything that they believed could improve their initial CSP formulations. They then interpreted this information as constraints and added them to their starting formulations. By using this method, they managed to achieve a significant reduction in the time it took their solvers to find solutions to many of their initial problems.

1.2 Hypothesis

The approach adopted in [2] was only semi-automatic. Once configured, the discovery of new information by HR is fully automatic , but there were many areas where the process would benefit from full automation. We performed an analysis of the full process and believed that all aspects of the process, that were previously manual, could be fully automated.

One of the main aspects of the HR configuration process is obtaining small examples of solutions to the problem. We believed this process could be automated by transferring the output from the solutions to small problems directly to the HR configuration files.

The process involved interpreting first order logic statements as constraints. For example, in order to demonstrate an improvement in the efficiency of a formulation, any new information must be translated into constraints in the syntax of the constraint solver (see §2.2). This was a manual task in the original process. We believed it would be possible to write a set of bespoke routines, based upon a Definite Clause Grammar, to convert first order logic statements into constraints automatically. We believed that this would be useful, not only in creating reformulations, but also in simplifying the creation of initial formulations for a new domain of investigation.

HR often produces a great deal of information, most of which will not be of any particular use in reformulating CSPs. It is a skilled task to be able to identify, within this large amount of data, that small sub-set which represents useful constraints. We believed that we could identify this sub-set much more easily if we automated the assessment and testing process.

In summary, by automating these parts of the process, we could reduce the amount of time and manual intervention the process required, thereby improving its effectiveness. In addition, we believed we could reduce the amount of expertise required by the user, in both constraint programming and the domain of investigation , for example Mathematics. This would make the process more accessible to less sophisticated users.

We built an application to implement a fully automated system. The application is responsible for controlling all aspects of the automated process in both interactive and batch modes. It provides an interface to a Prolog processor, which is used in the translation of theorems and testing of CSP formulations.

In order to confirm our hypothesis, we used the system to perform a number of investigations into various domains within finite algebra which are discussed in §2.8. In ten out of the eleven domains we investigated our system successfully identified theorems, in the output of HR, which reduced the time required by our initial solver to complete a full search for solutions. In addition, we identified a number of interesting mathematical properties of those finite algebras. These successful results lead to the conclusion that our initial hypothesis is correct and it is possible to fully automate the process.

1.3 This Document

The structure of this document is as follows:

- chapter 2 provides background to our work which the reader may find useful.
- chapter 3 outlines our review of the original approach and describes our automated system together with the decisions we took in developing it.
- chapter 4 provides more detailed information about the process we developed to translate first order logic statements into constraints.
- chapter 5 gives more details of the technical aspects of the implementation.
- chapter 6 describes the set up of experiments we performed in different domains.
- chapter 7 outlines the results of our experiments.
- chapter 8 comprises an analysis of the experimental results.
- chapter 9 contains a details of our conclusions.
- chapter 10 is where we suggest further work to be performed.
- appendix A contains the user manual for our application.

Chapter 2

Background

In this chapter we provide background information that the reader may find useful. We give more information about CSPs in §2.1. In §2.2, we describe some of the tools used for solving them and in §2.3 we outline some of the challenges that CSPs pose. We discuss machine learning in §2.4 and, in particular, the HR system in §2.5. The original semi-automated method is described in more detail in §2.6. Definite Clause Grammars, which we use to translate theorems, are discussed in §2.7 and the algebraic domains we have studied are outlined in §2.8.

2.1 Constraint Satisfaction Problems

A constraint satisfaction problem (CSP) consists of the following components:

- A set of variables $\{x_1, x_2, \ldots, x_n\}$
- A set of domains of values the variables can take
- A set of constraints specifying which values the variables can take simultaneously

A solution to a CSP is an assignment of values to each of the variables from their domains such that none of the constraints are broken. A large number of problems can be represented in this way and the solution of CSPs has applications in many areas, such as Mathematics, Electronics, Commerce and Scheduling. CSPs can be extremely complicated, involving a large number of variables and potentially complex constraints. CSPs are an extremely important area for research because they can be very time-consuming to solve. Methods are constantly being sought to improve the effectiveness of approaches used for solving them.

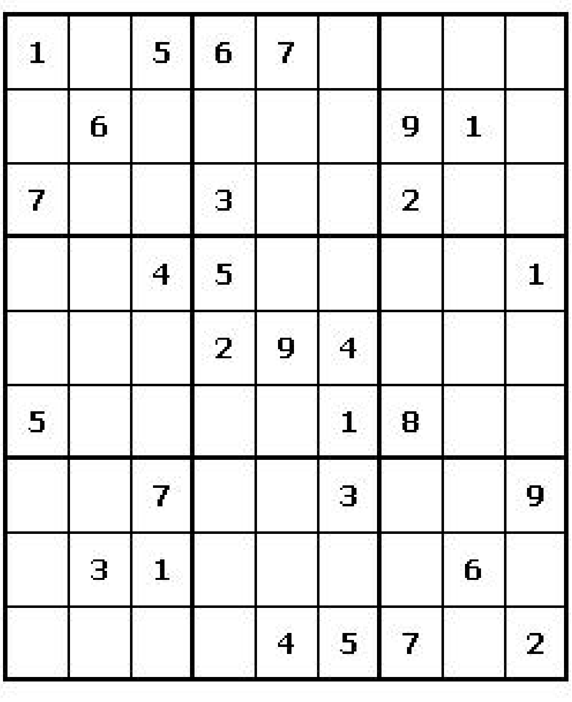
Here is a commonly seen example of a CSP problem:

$$\begin{array}{cccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

In this problem every letter represents one of the digits from 0 to 9, except S and M which must be non-zero. The problem can be specified as

variables	[S,E,N,D,M,O,R,Y]
domains	[S,E,N,D,M,O,R,Y]:[0..9]
constraints	$S \neq 0$
	$M \neq 0$
	S*1000 + E*100 + N*10 + D +
	M*1000 + O*100 + R*10 + E =
	M*10000 + O*1000 + N*100 + E*10 +Y

The problem is to find values for all of the variables that satisfy the constraints. Another commonly seen example of a CSP is the Su Doku puzzle where each square should be filled with a number from 1 to 9 with the numbers of each row, column and sub-square being different.



1		5	6	7				
	6					9	1	
7			3			2		
		4	5					1
			2	9	4			
5					1	8		
		7			3			9
	3	1					6	
				4	5	7		2

```
:-use_module(library(clpfd)).
qg(CT):-
        CT = [A1,A2,A3,B1,B2,B3,C1,C2,C3],
        domain([A1,A2,A3,B1,B2,B3,C1,C2,C3],1,3),
        all_different([A1,A2,A3]),
        all_different([B1,B2,B3]),
        all_different([C1,C2,C3]),
        all_different([A1,B1,C1]),
        all_different([A2,B2,C2]),
        all_different([A3,B3,C3]),
        labeling([],[A1,A2,A3,B1,B2,B3,C1,C2,C3]).
```

Figure 2.1: An example CSP formulation

2.2 CSP Solving Systems

A CSP solving system is a software tool that can be used to solve CSP problems. It allows users to declare variables, together with their domains and post constraints on those variables. Such solvers would then search for a solution to the problem by considering different possible values for the variables and confirming whether these assignments satisfy the given constraints. In modern solvers, this search is usually intelligently guided. For example, solvers normally apply methods, such as *forward checking*, whereby the solver assesses the impact of assigning a value to a variable by considering the effect of that assignment on the values that the other variables can take. Many solvers allow the user to specify different parameters to guide the search. An example of this is where a user suggests *fail-first* assignment as opposed to *lexicographical* variable assignment. *Fail-first* means that the solver will try assigning values to variables with the smallest remaining number of potential values rather than just assigning in the order the variables were originally declared.

There are a number of CSP solver tools available. Examples include ILOG JSolver, which is a commercial solver package that integrates with the Java programming language, Sicstus' Constraint Logic Programming over Finite Domains library and the Choco solver [5].

An example of a CSP solver formulation for Sicstus Prolog is shown in figure 2.1. This formulation is capable of finding quasigroups, which are described in §2.8 of size 3.
It has the following elements:

- *use_module(library(clpfd))* This instructs Sicstus to use the Constraint Logic Programming over Finite Domains (CLPFD) library. This library, which provides functions for solving CSPs involving variables with finite domains is included with the Sicstus distribution.

- *qg(CT)* This is the main predicate (or goal) of the solver formulation, which is queried to seek a solution. In the example, the goal is qg(CT) which looks for quasigroups of size 3. When a solution has been found, CT will be bound to the multiplication table of the solution.

- *CT=[...]* This is a variable declaration, which redefines the parameters of the goal predicate into the variables that the internal solver will actually work with. In the example, the multiplication table is redefined as a list of 9 named variables, A1 to C3, each representing a different element in the 3×3 multiplication table, where the letters in the variable name represent rows and the numbers are columns.

- *domain(...)* This indicates the values that the variables can take. In our case, all variables can take any value from 1 to 3.

- *all_different(...)* These are constraints on the variables. In our example, we know a quasigroup multiplication table is a *Latin Square*, where the values in the rows and columns are all different. The most efficient way, in terms of solving speed, of specifying this to a constraint solver is by posting an all_different constraint on the variables of each row and column. This is a built-in constraint of the Sicstus CLPFD library and is common to all modern solvers.

- *Labeling(...)* This declaration tells the solver to search and instantiate the problem variables to look for solutions to the CSP. The labeling predicate allows parameters to be passed to indicate that the search should use a particular variable ordering, value ordering or forward checking approach. In the example, all variables are to be found, using the default solving approach.

The method of reformulation that we apply in our system involves adding self-contained constraint statements to formulations similar to those shown above. These new constraint sets can be placed anywhere between the domain declarations and the final labeling declaration. Their placement is important because, during the search for a solution, constraints are triggered by the solver in the order they are declared . Hence, the impact of a constraint can change depending on where it is declared in the solver formulation. Two common methods of assessing the CSP solver performance are absolute CPU time taken and *backtracks*, a measure of the number of dead-ends the solver encountered.

2.2.1 Reification

Our approach makes use of reification, which refers to the process of treating a logical relationship as a concrete variable. The applicability of a particular constraint is linked to the value of a variable, where the variable is given the value 1 if the constraint applies and 0 if it does not. This mechanism underpins the ability of some CSP solvers to structure more complicated constraints.

Reification allows the solver search engine to dynamically post constraints as the solution search progresses, by linking the constraint to a reification variable that is then used in another constraint expression. In this manner, the application and triggering of constraints can be performed more effectively in both directions of the link.

In the case of Sicstus Prolog, the syntax of some operators, such as implications, requires that they be applied to reification variables rather than constraints themselves. Consequently, in order to correctly translate implication conjectures, it is necessary to apply reification syntax. This means that any constraints that are the operands of such operators must be reified. The syntax for expressing a reification of a constraint in Sicstus Prolog is as follows:

$$\textit{constraint}\ \#<=>\ \textit{variable}$$

2.3 CSP Solving Challenges

As mentioned in §1, various methods have been researched with the aim of improving the effectiveness of CSP solving. Two proven methods of improving the effectiveness of a CSP solver are *symmetry breaking* and *streamlining*

2.3.1 Symmetry Breaking

The n-queens problem is a standard CSP problem based upon the game of chess. The aim is to place n queens on an $n \times n$ chess board such that none of the queens attack any other. If we are searching for all solutions to this problem we can restrict the number of solutions we have to find by considering symmetries of the board. We know that, for each solution we find, there are a number of others that can be found, quite simply, by reflecting or rotating the original solutions. If we can restrict our search to only one example of a class of solutions then we can reduce the search space we need to cover. Such a simplification of the problem is known as *symmetry breaking.* A simple approach to solving the n-queens problem would be to place each queen consecutively in one of the $n \times n$ squares on the board. Another way of considering symmetry breaking is to look at the choices that may be made while finding a solution. By considering the rules of chess we can deduce that, with n queens to place, we will have to put each one in a different column and row. This reduces the problem to working through the rows and deciding which column to place the next queen. Every time we place a queen, we are now just choosing between, at most, n squares rather than $n \times n$ squares. The efficiency benefits that symmetry breaking brings makes identifying symmetries very valuable.

2.3.2 Streamlining

Decision making can be made easier if we rule out some of the possibilities and restrict ourselves to only a sub-set of the options. This applies to CSP solving.

If we can decide in advance which particular sub-set we are interested in, it is possible to get a satisfactory solution sooner. Such refinement of the problem search space is known as *streamlining.* It has been applied in the search for quasigroups (see §2.8) where we can speed up the search for solutions if we restrict ourselves to just idempotent examples, i.e. $\forall\, a\ (a * a = a)$. In [4], Gomes and Sellman showed how, by capturing regularities in subsets of low-order solutions to problems, they could *streamline* their search for examples of higher order examples. Smith et al [9] successfully applied streamlining to existence problems of spatially balanced Latin Squares, where they found solutions of order 35, significantly beating the previous best of size 18.

2.3.3 Reformulation using New Information

If we can find something new about the domain being studied then we may be able to use this knowledge to amend the CSP formulation we are using and improve the effectiveness of our solver. In [2], Colton et al. applied automated theory formation (ATF) techniques to identify theorems for a particular domain and interpreted them as reformulations to a basic CSP formulation. They showed how this new information, when properly integrated, could improve the effectiveness of their original approach.

2.3.4 Simplifying the Formulation Process

Much of the work involved in solving a particular CSP occurs at the formulation stage. The specification languages used in many CSP solving packages are complex and a great deal of work is required to take a high level problem and convert it into a good formulation from which the solver can provide an answer. The CONJURE system [3], has been developed by Frisch et al as a solution to the problem of converting high-level problem specifications into constraint programs automatically.

2.4 Machine Learning

Machine learning refers to the process of using computers to find knowledge about a particular problem domain. There are two approaches to this, *predictive* machine learning and *descriptive* machine learning. In *predictive* machine learning, we are interested in finding a function which, for a given example, can accurately predict the value for one of its characteristics based upon its other attributes. For instance, in predictive toxicology we want to be able to predict the toxicity or otherwise of a compound based upon its other characteristics, such as the number of benzene rings it has in its molecule. In order to generate a predictive function, we provide the system with a large set of examples called a training set. A *predictive* machine learning system applies a learning procedure to these examples in order to generate a predictive function, which could, for example, classify drugs as toxic or not when given values for different attributes. In *predictive* machine learning we are searching for a well-defined piece of information. By contrast, *descriptive* machine learning refers to the process of using a computer to speculatively find interesting pieces of information of relevance to the domain being investigated. Here, the idea is to present a general framework of a domain to the computer, provide a general idea of what it is we're looking for, by defining what would be interesting and configure the computer to search in certain ways. Our study aims to apply descriptive machine learning to the field of CSP solving. For a particular domain of problems, we ask the computer to find concepts and theorems, we assess those new theorems and consider their value in reformulating CSP formulations with a view to improving the effectiveness of the solvers we're currently using. Our approach uses the HR machine learning system, which is described in the next section.

2.5 The HR Machine Learning System

HR [1] is a machine learning system capable of generating new concepts about a domain of investigation and formulating proven conjectures about those concepts. In the mode of operation that we use, HR starts with a basic axiomisation of a particular domain, some examples of objects which satisfy those axioms and some basic concepts of that domain. All further concepts and conjectures that HR discovers are built from this starting set of concepts. HR's learning abilities have been applied to a number of diverse areas such as Mathematics, including Group Theory, Graph Theory and Number Theory, Bioinformatics and Music.

2.5.1 Concept Formation

New concepts are discovered by HR by passing existing concepts through a number of production rules. An example of a production rule is the *compose* production rule which takes the conjunction of two existing concepts to give the concept of objects which satisfy both of those concepts. Consider the concept of being red in colour and the concept of being a hat, this production rule would invent the concept of being both, i.e. a red hat. Similarly, the *negate* production rule would invent the concept of a hat which isn't red. HR uses a heuristically driven search to determine which production rules or concepts to use next and checks all formed concepts to see if they have been seen before.

2.5.2 Conjecture Formation and Proving

When HR has found a new concept, it considers how the concept applies to the examples it has been given. If the example set to which the concept applies is the same as that for a previous concept, then HR will make an equivalence conjecture between those concepts. This may include concepts which are true for all examples, in which case HR will make the conjecture that this concept is true for all members of the domain. If the concept is not true for any examples, then HR will conjecture that it is inconsistent with the axioms. In other cases, HR will search for conjectures linking concepts where their respective examples are subsets of each other. HR makes use of third party software. In particular, it attempts to prove conjectures by passing them to the Otter theorem prover [7] for proving and any that Otter fails to prove are given to the MACE [8] model generator to find a counterexample. This process produces implication conjectures. HR further seeks to refine these implication conjectures by considering conjectures where the consequent is the same as the original but the antecedent is a subset of the conjuncts that make up the original antecedent. Implication conjectures where the implication is not true for any proper sub-set of the conjuncts of the antecedent are known as prime implicates. These can be of greater value if they are stronger and less complex that the implication conjectures from which they were generated.

2.5.3 HR Configuration

HR is highly configurable, allowing the user to guide its search for concepts and conjectures in many ways. In order to investigate a problem area, the HR system requires a number of configuration files and some examples from the domain being investigated. This is provided to HR in two configuration files:

- *The Domain File* contains the core concepts required to axiomatise the domain and the examples of the domain that HR will use for its investigation.
- *The Macro File* is written in the HR scripting language and controls the production rule run process. This file configures HR for the particular discovery run, such as setting a parameter for the number of steps HR should undertake before stopping. In addition, this file also contains details of the axioms of the domain, which will be used by Otter to prove conjectures.

2.5.4 Output

HR can be configured to output its findings in a particular format to a particular location. It can also be configured to output only findings of a particular types, such as prime implicates only. This can be controlled via a report file and the macro. Example HR output, in the format we use is shown in figure 2.2.

```
all b c (((b*b=c & -(c*c=c)) -> (-(b*b=b))))
all b c (((b*b=c & -(b*b=b)) -> (-(c*c=c))))
all b c d (((b*c=d & c*b=d & -(d*d=d)) -> (-(c*c=c))))
all b c d (((b*c=d & c*b=d & -(c*c=c)) -> (-(d*d=d))))
all b c d (((b*c=d & c*b=d & -(d*d=d)) -> (-(b*b=b))))
all b c d (((b*c=d & c*b=d & -(b*b=b)) -> (-(d*d=d))))
all b c ((((exists d (d*b=c))) -> ((exists e (e*e=b)))))
all b c (((exists d (d*d=b))))
all b c ((((exists d (d*b=c))) -> ((exists e f (b*b=e & f*e=c)))))
all b c (((exists d e (b*b=d & e*d=c))))
all b ((((exists c (b*b=c & c*c=c))) -> (b*b=b)))
all b c ((((exists d (b*c=d & c*b=d))) -> ((exists e (b*b=e & e*e=c)))))
all b c ((((exists d (d*d=b & d*d=c))) -> ((exists e (b*c=e & c*b=e)))))
all b ((((exists c (c*b=c))) -> ((exists d e (d*e=b & e*b=e)))))
all b c ((((exists d (b*c=d & c*b=d))) -> ((exists e (b*e=c & c*e=c)))))
```

Figure 2.2: Example output from HR

2.6 A Semi-automated Approach to CSP Reformulation

In [2], HR was used to generate implied and induced constraints to be used in reformulating CSP formulations in order to improve the effectiveness of solvers. The method presented there consisted of 4 phases, which are shown in figure 2.3.

- *Phase One* A domain of investigation was chosen and a basic solver formulation was developed to produce examples of elements of the domain.
- *Phase Two* HR was configured with the axioms and concepts of the domain. In addition, HR was given examples of the domain, found by solving the basic formulation. HR then ran for a specified period to discover proved theorems about the domain and its concepts, using Otter to prove the theorems.
- *Phase Three* The results from HR were reviewed to identify which theorems could be useful as constraints. This resulted in a small sub-set of HR's output which were interpreted as constraints and added to the basic solver formulation.
- *Phase Four* The reformulations were tested and compared to the original solver formulation. Those that performed the best were used in attempts to solve larger instances of the problem.

This method achieved some success. They identified a number of useful constraint-forming conjectures within the HR output. They demonstrated how,

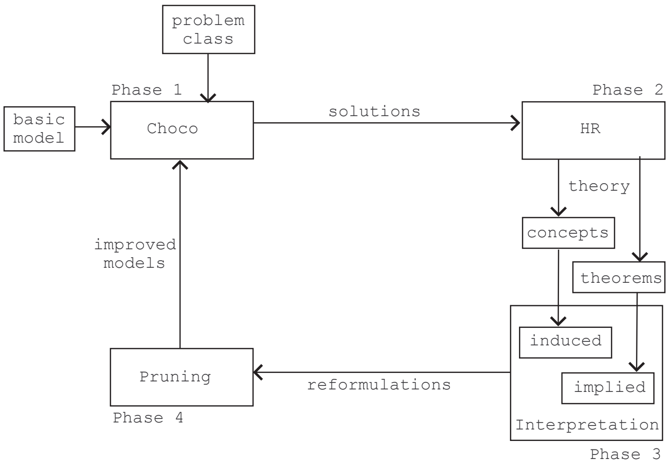


Figure 2.3: A Semi-automated Approach to CSP Reformulation [2]

by converting them to constraints, they could improve the efficiency of their CSP solvers. In particular, they showed how the time spent finding constraints and reformulating the solvers is compensated by an improvement in the efficiency of the solver formulation. This demonstrated how the combination of AI techniques can lead to improvements in the performance of at least one of those techniques.

2.7 Definite Clause Grammars

Definite Clause Grammars (DCGs) are sets of of rules for parsing and reformulating phrases. DCGs specify how a given phrase should be interpreted by defining the forms that expressions and sub-expressions can take. For example, a DCG for parsing simple English sentences is shown in figure 2.4. This could be used, for instance, to determine whether a given sentence is valid.

DCGs are useful for taking the key information from a given phrase and reformulating it in a standard manner. We use DCGs to convert textual first order theorems into a form which is more usable by identifying keywords, operators and variables and re-stating them in a standard format. For example, our DCG could take a phrase of the form "all a (. . .)" and extract the basic format of the expression as being a universally quantified expression over the variable a.

Sicstus Prolog, in common with many Prolog distributions, provides a DCG interpreter. The general syntax for a DCG phrase is:

Head --> body

```
sentence --> noun_phrase, verb_phrase.
noun_phrase --> article, noun.
verb_phrase --> verb, noun_phrase.
article --> [the].
article --> [a].
noun --> [man].
noun --> [woman].
verb --> [loves].
verb --> [hates].
```

Figure 2.4: An Example DCG for Simple English Sentences

which states "a possible form for 'head' is 'body'". When a phrase is given to the parser for conversion, the prolog interpreter compares the format of the phrase with the parsing rules it has been given.

In order to use our DCG rules, we have to express the input as a list of string tokens, for which we have used bespoke tokenizer predicates. This is because the interpreter uses *difference lists*, where lists are considered to be in the form

$$\text{list} = [\textit{something}] + [\textit{rest\_of\_list}]$$

allowing the processor to deal with the *something* without being necessarily concerned with the form of *rest_of_list*. An understanding of difference list processing is vital to creating a usable DCG. It is also possible to include normal Prolog statements within the DCG rules in order to tailor the output.

2.8 Algebraic Domains

Quasigroups

A quasigroup is a finite algebra, i.e. a set of mathematical objects, with a binary operator $* : Q \times Q \rightarrow Q$ such that $\forall\ a\ b\ \exists\ x\ y\ (a * x = b \land y * a = b)$. Quasigroups of all sizes can be found, however there are open questions regarding the existence of specific types of quasigroups of various sizes. These specific types of quasigroups each have additional axioms, shown in figure 2.5.

Quasigroup existence problems fit neatly into the realm of CSPs by considering, as problem variables, the multiplication table showing the results of the binary operator on different pairs of elements. For a quasigroup of size n, we therefore have n^2 problem variables, each representing a solution to one of a*b where $a, b \in Q$. An example quasigroup, of size 4 is shown in figure 2.6. We define the domains of the problem variables by labelling the n elements of the algebra using the natural numbers $\{1, \ldots, n\}$, each of the results of the binary operator, i.e. the problem variables, must take a value in $\{1, \ldots, n\}$. The additional quasigroup axioms can be interpreted as constraints along the rows

QG1: $\forall\ u\ v\ w\ x\ y\ z\ (x * y = u \wedge z * w = u \wedge v * y = x \wedge v * w = z) \rightarrow (x = z \wedge y = w)$
QG2: $\forall\ u\ v\ w\ x\ y\ z\ (x * y = u \wedge z * w = u \wedge v * x = y \wedge v * z = w) \rightarrow (x = z \wedge y = w)$
QG3: $\forall\ a\ b\ (a * b) * (b * a) = a$
QG4: $\forall\ a\ b\ (a * b) * (b * a) = b$
QG5: $\forall\ a\ b\ ((a * b) * a) * a = b$
QG6: $\forall\ a\ b\ (a * b) * b = a * (a * b)$
QG7: $\forall\ a\ b\ (b * a) * b = a * (b * a)$

Figure 2.5: Quasigroup Subtype Axioms

*	0	1	2	3
0	0	1	2	3
1	1	2	3	0
2	2	3	0	1
3	3	0	1	2

Figure 2.6: Size 4 Quasigroup

and columns of the multiplication table. Structurally, it states that, for a given element, every other element must appear at least once as the result of a multiplication with the original element both as the left or right multiplicand. In other words, each row and each column of the multiplication table must contain all elements, we say that the multiplication table is a *Latin Square*. The search for quasigroup sub-types is relatively straightforward for small examples. However, the processing time required to find larger examples increases significantly. Consequently, any improvements in the formulations to find small examples should, when applied to looking for larger examples, represent a significant time-saving. Quasigroups appear in various scientific areas such as Cryptography and, in addition, the Su Doku puzzle shown earlier is an example of a size 9 quasigroup.

Loops

Loops are quasigroups with the additional axiom that the set contains an identity element, i.e. $\exists e\ \forall a e * a = a * e = a$. An example of a size 4 loop is shown in figure 2.7. In this example, the identity element is 1.

*	0	1	2	3
0	3	0	1	2
1	0	1	2	3
2	1	2	3	0
3	2	3	0	1

Figure 2.7: A Size 4 Loop

Semigroups and Monoids

Semi-groups are sets with an associative binary operator i.e. $\forall a\ b\ c\ (a * b) * c = a*(b*c)$. A semigroup need not have an identity or inverse element and therefore does not, necessarily, have to be a quasigroup. An example, size 4, semi-group is shown in figure 2.8. Monoids are semi-groups that also have an identity element.

*	0	1	2	3
0	2	1	3	1
1	1	2	3	2
2	3	3	3	3
3	1	2	3	2

Figure 2.8: A Size 4 Semi-group

Groups

Groups are monoids where every element of the group has an inverse element i.e. $\forall a\ \exists b\ (a * b = b * a = e)$. Group Theory, which covers the study of all these finite algebras, is an important area of mathematics and has many applications in Mathematics and other scientific fields such as Chemistry and Quantum Mechanics.

Isomorphism

Algebraic structures are said to be isomorphic if there is a one to one mapping between their respective multiplication tables. By this we mean that the multiplication table for one could be obtained from the other simply by swapping the labels of the elements. Isomorphism is indicated, mathematically, by the symbol $\cong$. The two tables shown in figure 2.9 are isomorphic. Table B can be obtained from Table A by simply relabeling the elements of the table such that 1 becomes 2, 0 becomes 1 and 2 becomes 0.

*	0	1	2	3
0	1	1	1	1
1	0	0	0	0
2	3	3	3	3
3	2	2	2	2

(a) Table A

*	0	1	2	3
0	2	2	2	2
1	1	1	1	1
2	3	3	3	3
3	0	0	0	0

(b) Table B

Figure 2.9: Isomorphic Tables

Chapter 3

Design Decisions

In the previous chapter we discussed background information useful to the understanding of the project. We now consider the design decisions we took in developing the fully automated system. In section 3.1 we perform a review of the Semi-automated approach outlined in §2.6 to consider where potential improvements could be made. In §3.2 we discuss the high level design of our system, ICaRuS.

3.1 Semi-Automated Approach Review

The method applied in [2] was very successful. They identified a number of useful constraint-forming conjectures within the HR output. They demonstrated how, by converting them to constraints, they could improve the efficiency of their CSP solvers. In particular, they showed how the time spent finding constraints and reformulating the solvers is compensated by an improvement in the efficiency of the solver formulation. We consider here ways to further improve the efficiency of that method. In the paragraphs below, we describe some of the areas where we believe improvements in the process can be made and in §3.2 we discuss the system that we have developed to apply these improvements.

Preparing Initial Formulations

CSP programming is a skilled task and the process of creating the initial formulations can be time consuming. In [2], this task was performed manually using the Choco [5] solver. We describe here a method by which we can formulate the initial CSP formulations in an easier and quicker manner.

Configuring HR

In [2] the HR configuration was performed manually. Our approach includes a method of automating this process to reduce set-up and overall run-time.

Reviewing HR's Output

Given a domain of investigation, HR produces a large number, in some cases thousands, of conjectures. Only a very small sub-set of these conjectures are potentially useful as constraints. In order to identify these constraints, it is necessary to review all the HR output. In [2] this review was performed manually. We have developed a method of assessing HR's output automatically, removing this manual task.

Translating theorems and Reformulating CSPs

Converting HR's output, once reviewed, into usable constraints was again a manual task. We considered that a system to automatically convert the conjectures into constraints and reformulate CSP formulations would be very useful in improving the overall effectiveness of the approach.

Removing the Need For Specialist Skills

Two key areas of the method applied in [2] required specialist skills:

- *constraint programming skills* were required in producing the initial formulations which formed the starting point for the investigation, and
- *domain specific skills*, namely Mathematics expertise, were required to review the output from HR.

The approach we outline here aims to significantly reduce the need for such expertise.

3.2 An Integrated Constraint Reformulation System (ICaRuS)

In order to implement the improvements to the method of [2], we identified in §3.1, we developed an integrated system, ICaRuS. The workings of the system are described in the following sections. The system is outlined diagrammatically in figure 3.1

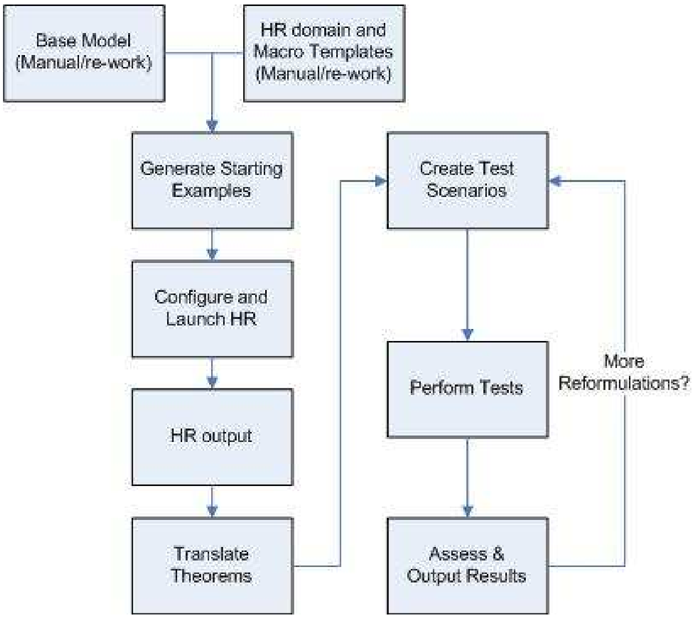


Figure 3.1: Process Overview

3.2.1 Starting Solver Specification

As in [2], the process begins with a basic CSP solver formulation. This is the simplest formulation to naively solve the given problem and is formulated using just the basic axioms of the domain being investigated. It is used to generate example solutions to the CSP that will be provided as examples to HR. For instance, in the case of quasigroups, they would be examples of quasigroups of a given size. The system we have developed represents each solver formulation as a list of formulation elements, equivalent to each of the various elements shown in the example formulation in §2.2, i.e. domain declarations, variable declarations etc. The list of elements contains all the information required by the system to create the prolog code for a basic solver formulation. Each element either:

- indicates prolog code that should be run to generate part of the formulation. For example it could indicate a predicate *declare(N)* which would write the part of the prolog formulation responsible for declaring N variables, *or*

- indicates a first order logic statement that should be translated into constraints using the theorem translation process described in chapter 4.

Users can create basic formulations by adding to and amending elements of the list. For instance, by using a template from another similar domain and adjusting or adding elements, a user can significantly reduce the set-up cost for running the experiment. A user who is not familiar with constraint programming but who is familiar with first order logic may choose to state any additional constraints simply in first order logic and have the system automatically generate the underlying constraints. For instance, this was very useful in moving the domain of investigation from quasigroups to loops. By simply adding the identity element constraint in first order logic to the basic formulation template for quasigroups, we had a basic formulation for loops. To reduce potential communication issues, the system has adopted the first order logic format understood by Otter ("Otter format").

3.2.2 HR Configuration

The system provides a method whereby both types of HR configuration files (see §2.5) may be configured and generated automatically. The system will generate examples from the base formulation and add them to the domain file. It will also add any axioms of the domain to the macro file as necessary. The domain axioms are provided in Otter format, so the user can use the same axioms to configure HR and create the basic formulations. HR's starting concepts are explicitly stated in a domain file template.

This part of the process generates the configuration files from templates containing tags indicating where information is to be placed. Again, the system allows users to base any new configuration upon old configuration files that were used for another domain, thus reducing set-up time. The launching of the HR application is controlled by the system. The location of the HR output will be specified in the macro template - which can be manually updated as necessary.

3.2.3 Theorem Translation and Assessment

In [2], the theorems output by HR were individually reviewed and assessed to see if they had properties which would make them valuable constraints. Any potentially useful theorems were then translated into constraints. In the automated system, this process is reversed. All theorems that HR produces are translated into constraints initially. They are then assessed for effectiveness by adding them to the base solver formulation and measuring any improvement in the solver's performance. This approach increases translation workload and may mean that many uninteresting theorems are converted and tested. However, this approach removes any need for domain specific (e.g. mathematical) knowledge, it potentially improves coverage of the HR output data and may mean that some interesting theorems may be identified which may otherwise have been missed. Each of the theorems that HR produces is translated into constraints using the translation process described in detail in chapter 4.

3.2.4 Assessment of Theorems

We are only interested in theorems which improve the efficiency of a constraint solver formulation. A simple generate/test model has been developed for assessing whether a theorem is useful. The benchmark for the performance test is the time the basic solver formulation needs to find all possible solutions for a given domain and size. Finding a single solution or first n solutions is inappropriate as this will be influenced by the distribution of solutions within the search space. Hence, an exhaustion of the search space is used. This approach is still valid if there are no solutions as better constraints will improve the solver's ability to search all possibilities and exhaust the search space. Once a theorem has been translated, it is introduced into the basic solver formulation and the reformulation is re-tested. This testing normally involves calling a wrapper predicate which runs the formulation as many times as are required to exhaust the search space and returns statistics about the run. If the reformulation exhausts the search space more quickly, then the formulation is adjudged to be more effective.

The theorems output by HR are translated into constraints. The translation process produces sets of constraints that are self-contained and can easily be introduced into a formulation by placing them at an appropriate place among the existing formulation statements. The placement of constraints into the formulation is important in the performance of a test formulation. Therefore all possible placements of the translated theorem are considered. In setting up a base formulation for testing, the user must indicate which parts of the solver formulation are movable, to allow constraints to be added between them.

3.2.5 Testing Rounds

Before the first round of testing is performed, the system prepares all possible reformulations using the base formulation and HR's theorems. Each round of testing consists of running each new reformulation in turn and recording the time taken to exhaust the search space. The testing process applies a genetic algorithm approach whereby a certain sub-set of the population of reformulations "survives" to take part in the next generation of testing. All surviving reformulations take part in "reproduction", taking the theorems used by other reformulations and using them to make new reformulations. The user can specify the cut-off level for survival of a reformulation into the next round by stating a proportion of the time the base formulation, or alternatively the best formulation so far, takes to exhaust the search space. Any reformulations that perform better than this cut-off are included in the next round. The user can specify whether the "offspring" will contain all theorems in HR's output or just theorems being used in surviving reformulations. Testing ends when there are no more possible reformulations using acceptable theorems or when the reformulation limit, i.e. the number of theorems to add to a formulation, has been reached.

For example, if the HR output file contains three theorems and the base formulation has two spaces to place new constraints then there will be six reformulations in the first round. Any that survive to the next round may be reformulated using theorems that they didn't use in the first round. This will continue until all tests have used all theorems or the limit on adding theorems has been reached.

§A.2.4 describes how the testing parameters can be set when using the GUI application and provides some more detail of the individual parameters.

Results and Output

The system can either present the results of the test run to the user visually or save the results in a report file for them to review. This gives details of the performance of reformulations and how they were created, together with statistics about the processing run.

3.3 Sicstus Prolog

In determining which solver system to use in the project, we considered a number of options. Amongst the candidates were the MACE model generator and the Choco solver [5], which were used to generate models in [2]. In addition, we considered using the ILOG JSolver, a commercial java-based solving library and the Sicstus Prolog package, which is distributed with it's own built-in solving library.

The Sicstus Prolog solver is a fully functioned modern solver. In simple tests, the solver out-performed MACE and easily beat previous Choco results and after this initial period of familiarisation we considered that Sicstus would be suitable. The CSP solving element of Sicstus Prolog is the Constraint Logic Programming over Finite Domains (CLPFD) library [6].

3.3.1 Prolog Interface

Prolog is used to perform the translation of theorems, to generate solver formulation Prolog source and to test solver formulations. It is possible to create user interfaces in Sicstus Prolog using the TCL/TK library and this was investigated. In addition, it is possible to write procedural Prolog code which can automate repetitive tasks. This would have resulted in an application written entirely in Prolog, which would have removed the need for an interface between languages. However Java was considered to be most appropriate for building the application GUI. It is very flexible, allows for easy development of good quality GUIs. It is much simpler to automate tasks using Java. It is more portable, does not require additional installations and involved a shorter learning curve.

However, this meant that some method of interfacing Java with Sicstus Prolog had to be developed.

The Sicstus Prolog application supports Java and Sicstus communication in three main ways. These are:

- the Jasper java library
- the Prologbeans library
- via sockets

The Jasper library promised to be the most suitable, allowing the developer full control over starting and stopping a Prolog server. However, initial investigation into using Jasper proved unsuccessful due to installation problems on the Windows XP machine we initially used. The initial problems with Jasper were recently resolved following discussions with Sicstus staff and future development may see the application ported to Jasper.

Prologbeans is a library which uses sockets to connect Java to Sicstus Prolog. Therefore the developer does not need to work at the socket level. Prologbeans has some problems. In particular, it does not provide functions to start and stop a Prolog server, which means an alternative method of creating a background Sicstus process is required. In addition, its implementation of exception handling is not straightforward and the timeout function has issues which can leave the server hanging. However, workarounds have been found and implemented, see §5.

3.3.2 Generating Tests Every Time vs Once

Part of the overhead of running tests is in generating the reformulations. This involves calling prolog predicates to write the components to a prolog source file, which is then consulted. This is done repeatedly for each new reformulation and therefore includes repetition of formulation elements that are common to more than one formulation, such as those in the base formulation. This introduces some redundancy into the testing process. An alternative would be to generate the formulation elements once and store the results in memory or in local files to be brought into the reformulations as required. However, the memory cost of doing this would be significant and unmanageable. Many generated axioms are over 200kb in size. In addition, the most significant overhead in creating the macro files is not in generating the prolog, but in writing this output to persistent storage and this would not be avoided.

Chapter 4

Algorithmic Specifications

In this chapter we describe the theorem translation part of the system. The functions described here are capable of translating first order logic statements into constraints to be used in formulations for the constraint solver. This part of the system is used in translating any theorems from HR (see §2.5) into constraints and also to create starting formulations (see §2.2) when using first order logic to specify them. Figure 4.1 shows the theorem translation process.

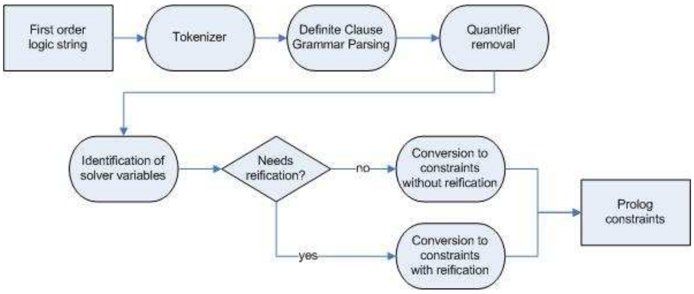

First order logic string
Tokenizer
Definite Clause Grammar Parsing
Quantifier removal
Identification of solver variables
Needs reification?
no
yes
Conversion to constraints without reification
Conversion to constraints with reification
Prolog constraints

Figure 4.1: Theorem Translation Overview

4.1 Overview

The translation of theorems is achieved using a collection of Prolog predicates. The main components of this are:

- a bespoke Definite Clause Grammar (DCG), which breaks a given theorem into a nested set of operators, *and*
- a quantifier removal routine which converts a universal or existential quantifier into a collection of possible individual cases.

The individual components of the translator are each defined in separate Sicstus Prolog modules which are called from a top level expand_axiom/1 predicate. The expand_axiom/2 predicate takes a given axiom and an options list and writes the resulting translation to standard output, which can be re-directed to any opened file, which is usually the current solver formulation file being generated by the system.

4.2 String Tokenizer and Token Identification

The first stage in converting a given theorem is to tokenize the string of characters into a list of tokens, removing any whitespace characters. A string presented to Sicstus Prolog is interpreted as a list of ASCII character codes. The tokenize/2 predicate unifies an ASCII character code list with its representation as a list of individual strings (ASCII code lists). The string is separated using knowledge of character codes for spaces, operators and brackets, and the characters of individual tokens are gathered together by analysing their ASCII codes. The token_list/2 predicate is then used to convert the list of individual strings into lists of atoms which identify the type of the axiom element, such as variable or number, and the element itself. This predicate makes use of predicates to identify strings as keywords, numbers or atoms.

For example, A tokenisation of this axiom

all a b ((a*b=b) → (exists c (c*b=a)))

would be as follows

[all,var(a),var(b),(,(,var(a),*,var(b),=,
var(b),),-,>,(,exists,var(c),(,var(c),*,
var(b),=,var(a),),),)]

4.3 Definite Clause Grammar

We developed a definite clause grammar (DCG) to parse a given tokenized theorem and convert it into a nested set of operators. The grammar encompasses aspects to recognise and convert such things as quantifiers, operators, variables, numbers and bracketed expressions. The grammar also includes a notion of precedence of operators by considering the sub-terms of lower order operators to be of the form of the results of higher precedence operations, forcing those to be evaluated first and nested deeper. The grammar is able to parse any first order Otter format theorem, included nested operations, such as $allab(((a*b)*b) = a)$ and could easily be adapted to include other operations such as the addition operator of ring theory. The grammar outputs nested expression terms, making them easier to process further. The example theorem from above is evaluated by the DCG as follows:-

all([var(a),var(b)],e(->,e(=,e(*,

```
var(a),var(b)),var(b)),exists([var(c)],
  e(=,e(*,var(c),var(b)),var(a)))))
```

It is useful to know the meanings of some of the component elements:

- *all(variable_list,expression)* - represents a universal quantification of *expression* over the variables listed in *variable_list*.
- *exists(variable_list,expression)* - represents an existential quantification of *expression* over the variables listed in *variable_list*.
- *e(op, sub1, sub2)* - represents the action of (binary) operator *op* on sub-expressions *sub1* and *sub2*.

In this example translation, the outermost expression is a universal quantification over the variables a and b. The next innermost expression is an implication expression, and so on. The difference list processing that underlies the DCG parsing recurses from left-to-right along the length of the list. However, this is not suitable for parsing theorems that may contain nested sub-expressions or whose terminal elements may need to be translated in a manner which incorporates the results of translations from earlier in the parsing tree. Consequently, the DCG was developed to simulate left-recursion by maintaining a record of previous parsing results when making a further recursive call.

4.4 Removal of Quantifiers

Once a theorem has been converted into a nested expression of operations, the simplest method of converting the theorem into constraints is to expand it to remove any quantifiers. By this, we mean that we restate the expression by considering all the specific cases for which the expression can hold. The basic idea is that universal quantifiers are expanded by considering them as conjunctions of individual necessary cases. Similarly, existential quantifiers are expanded as disjunctions of individual possible cases. For example, the theorem $\forall\ a\ (a*a = 1)$ can be expanded by stating that $1*1 = 1 \wedge 2*2 = 1 \wedge 3*3 = 1 \ldots$ etc. Also, $\exists\ a\ (a*a = 1)$ is expanded to say that $1*1 = 1 \vee 2*2 = 1 \vee 3*3 = 1$.

When there are multiple quantified variables, we risk producing a translation that is very large, covering a vast number of possible scenarios. The alternative would be to try to generate a smaller constraint statement that includes the notion of application over a certain set of variables. However, there is no difference, in terms of the number of constraints posted to the solver store, between making this explicit statement in a CSP solver formulation or in posting a general expression that covers multiple variable values. In addition, it is more difficult to generate Prolog constraint statements that cover generalities and results in the translation becoming increasingly domain specific.

The expansion of our example axiom from above for a domain of size 2 is

```
e(&,e(&,e(->,e(=,e(*,1,1),1),e(;,e(=,e(*,1,1),1),
e(=,e(*,2,1),1))),e(->,e(=,e(*,2,1),1),e(;,e(=,
e(*,1,1),2),e(=,e(*,2,1),2)))),e(&,e(->,e(=,e(*,1,2),2),
e(;,e(=,e(*,1,2),1),e(=,e(*,2,2),1))),e(->,
e(=,e(*,2,2),2),e(;,e(=,e(*,1,2),2),e(=,e(*,2,2),2)))))
```

Note the inclusion of a number of e(&,_,_) expressions representing a conjunction of various specific cases.

4.5 Identification of Solver Variables

The next stage of the process is a search through the expression to identify any sub-expressions which represent variables in the solver formulation and to replace them with those variables. This is performed recursively using the replace_constraint_variable/2 predicate. For example, in finite algebras modelled as multiplication tables, the solver formulation variables are the results of the * operator on pairs of variables. Therefore, we can replace any occurrences of the expression *e(*,n,m)* by the variable representing the result of $n * m$ (or the n^{th} row and m^{th} column variable). It isn't possible to replace all such expressions with variables from the formulation, for instance the result of (1*1)*(2*2) is the combination of two sub-results and cannot be simplified as a single solution variable. In our example axiom, when we have searched and replaced solver variables, our expression looks like this:

```
e(&,e(&,e(->,e(=,cvar(E_1_1),1),e(;,e(=,cvar(E_1_1),1)
,e(=,cvar(E_2_1),1))),e(->,e(=,cvar(E_2_1),1),e(;,
e(=,cvar(E_1_1),2),e(=,cvar(E_2_1),2)))),e(&,e(->,
e(=,cvar(E_1_2),2),e(;,e(=,cvar(E_1_2),1),e(=,
cvar(E_2_2),1))),e(->,e(=,cvar(E_2_2),2),e(;,
e(=,cvar(E_1_2),2),e(=,cvar(E_2_2),2)))))
```

where, for example, the expression e(*,1,1) has been replaced by the solver variable E_1_1 and enclosed with cvar(. . .) for easy identification later.

4.6 Converting to Constraints

The last stage of the translation process deals with converting the expanded expressions, in the form of *e(operator,sub1,sub2)* into constraints on the variables of the CSP solver formulation. This part of the process is somewhat system specific in that the constraints must obey the syntax of the underlying CSP solving system, in our case Sicstus Prolog. The most straightforward method of converting expressions to constraint statements is to write write to standard output, redirecting as necessary for output to persistent storage. This is achieved using the convert/1 predicate.

The recursive conversion routines process the expression, and output the resulting constraints as text to the current output stream. The results of subexpressions are returned from the recursion, for use in expressions nested at a higher level. In our example above, the converted axiom becomes:-

```
((((E_1_1#=1)->((E_1_1#=1);(E_2_1#=1))),((E_2_1#=1)->
((E_1_1#=2);(E_2_1#=2)))),(((E_1_2#=2)->((E_1_2#=1);
(E_2_2#=1))),((E_2_2#=2)->((E_1_2#=2);(E_2_2#=2))))),
```

Note that the solver variables in the form *cvar(...)* have been extracted directly. In addition, the numerous *e(&,sub1,sub2)* expressions are replaced, simply by the standard Prolog ',' conjunction operator. This is now in the form of a syntactically acceptable constraint on the variables of the solver formulation. The above statement contains implications and would not be syntactically acceptable to Sicstus prolog. Reification must be used to correctly translate this theorem. This is discussed in the following section.

4.6.1 Reification

In the case of Sicstus Prolog, operators must obey specific syntax rules and cannot simply be stated as first order clauses. In most cases, the expressions can be reduced to simple constraints on individual variables of the solution. However, some expressions, such as implication conjectures, can only be posted as implications upon reified constraints. This requires additional processing so, in the first instance, the convert/1 predicate determines, via a recursive review of the expression, whether it contains operators that would need reification. If so, reification is used, otherwise standard conversion is applied.

In Sicstus Prolog, the syntax *constraint* # <=> *variable* is used to establish a reification relationship between *constraint* and *variable*. A translation involving reification is achieved by amending the recursive translation predicates in such a manner that they apply reification syntax as necessary in expressions and pass those reified variables back to higher levels of the recursion for use in generating higher level expressions. Our example theorem, therefore, when correctly translated into constraints using reification becomes:

```
((((E_1_1#=1 #<=> V3),((E_1_1#=1 #<=> V5),
(E_2_1#=1 #<=> V6),V5 # V6 #<=> V4),V3#=>V4
#<=> V2),((E_2_1#=1 #<=> V8),((E_1_1#=2
#<=> V10),(E_2_1#=2 #<=> V11),V10 # V11
#<=> V9), V8#=>V9 #<=> V7),V2 #/\ V7#<=>V1),
(((E_1_2#=2 #<=> V14),((E_1_2#=1 #<=> V16),
(E_2_2#=1 #<=> V17),V16 # V17 #<=> V15),
V14#=>V15 #<=> V13),((E_2_2#=2 #<=> V19),
((E_1_2#=2 #<=> V21),(E_2_2#=2 #<=> V22),
V21 # V22 #<=> V20),V19#=>V20 #<=> V18),
V13 #/\ V18#<=>V12),V1 #/\ V12),
```

In this translation, the equality of E_1_1 with 1 is reified to the variable V3 which is used later as the antecedent in an implication conjecture. We developed a mechanism to provide an incremental list of unique reification variables to the translation process. This is performed with the reset_vars/0 and next_variable/1 predicates which make use of the built-in assert/1 and retract/1 predicates to update the Prolog database with the value of the next variable available.

Chapter 5

Implementation Details

In this chapter, we discuss details of the implementation of the ICaRuS application, the design of which is discussed in §3.2. Appendix A shows the user manual for this application.

5.1 Application Overview

The application is split into two main parts:

- a Java element that controls the process and handles interaction with the user, *and*
- and a Prolog element which handles translation of theorems and generation and testing of formulations.

The Prolog elements have been discussed in §3. Therefore, this section mainly covers the Java side of the application.

5.2 Class Structures

This section describes the java classes that have been developed to support the application.

5.2.1 Main Classes

These comprise the main application class, a JFrame, which displays the GUI, handles events and controls batch running. There is a file viewer JFrame for viewing the contents of files. In addition, there is a Config class which has the aim of centralising configuration variables, which it statically provides to all the other classes of the application.

5.2.2 Solver Formulation Classes

These classes are used to represent various elements of a formulation for solving CSPs. The basic UML class hierarchy is given in figure 5.1.

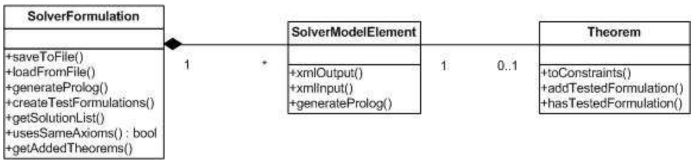


Figure 5.1: Basic UML: Solver Formulation Classes

SolverFormulation

SolverFormulation is the class that represents a CSP solver formulation - it contains an ArrayList of SolverFormulationElement objects, used to generate the Prolog source. This class also contains details of the predicate to be run to produce solutions in a format appropriate for HR configuration. This class includes the following methods:

- *saveToFile* Saves the solver formulation as XML to a specified file.
- *loadFromFile* parses XML to create a solverFormulation object.
- *generateProlog* generates Prolog source using the individual SolverFormulationElement
- *createTestFormulations* creates a list of reformulations by placing a given new theorem in various positions within the existing formulation.
- *getSolutionList* uses the formulation to generate solutions in HR input format using the solution predicate.
- *usesSameAxioms* compares two solverFormulations to evaluate which theorems they contain, used in generating new test scenarios.
- *getAddedTheorems* returns a list of the theorems being used by a solverFormulation, for test purposes.

SolverFormulationElement

SolverFormulationElement stores information about how one particular part of the prolog formulation is to be generated. A boolean value indicates whether the code will be generated from a first order logic statement, which it stores as a *Theorem*, or by running generator code, for which it stores a predicate and filename. Further boolean values indicate whether this element is movable and active. "Movable" affects the placement of new axioms in generating test scenarios and only "active" elements will form part of the generated Prolog source. This class includes the following methods:-

- *xmlOutput* Used for XML saving.
- *xmlInput* used for parsing solver XML.
- *generateProlog* generates Prolog source for the element using the interface to Sicstus Prolog.

Theorem

Class *Theorem* encapsulates a simple first order logic theorem, stored as a String. The class includes the *toConstraints* method which generates the constraint code. During testing a theorem records all the reformulations it has been added to so that it is not added twice to the same formulation, the *addTestedFormulation* and *hasTestedFormulation* methods are used here.

Other

Other classes include exceptions for handling failures when translating theorems and generating solutions and an extension of *AbstractTableModel* for displaying a *SolverFormulation* in a JTable.

5.2.3 HR Interface Classes

These classes handle creating the HR configuration files, and invoking an external HR application.

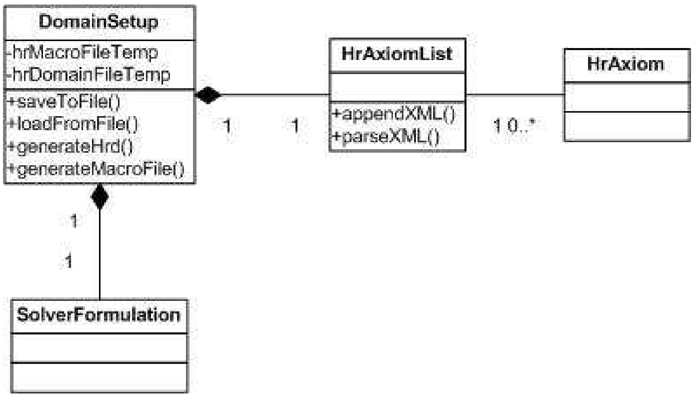


Figure 5.2: Basic UML: Domain Setup Class

Class *DomainSetup* encapsulates everything required to start up and run a full batch session. This includes details of a HR macro template, a HR domain template, a list of domain axioms and a solver formulation.

- *saveToFile* saves the domain setup in XML format.
- *loadFromFile* loads a previously saved set-up.
- *generateHrd* starts the process for generating a HR domain file using the template and base formulation to generate examples.
- *generateMacroFile* starts the process of generating the HR macro file.

The creation of the domain and macro files are handled by dedicated processing threads to improve GUI performance, see below.

- *HrAxiomList* embodies the list of axioms that underly the domain being investigated. It includes methods *appendXML* and *parseXML* to support loading and saving domain files.
- *HrAxiom* objects represent each of the axioms in the axiom list.
- *HrInterface* controls launching an external HR process in either batch or visual modes. It also provides statistics about the HR run time.
- *ThreadFileGenHrd* generates a HR domain configuration file from a template file. It first obtains solutions to problem from the base formulation and inserts them into the template. The template is then saved in a directory that HR uses to load configuration files.
- *ThreadFileGenMacro* generates a HR macro file from a template. It inserts the axioms of the domain and a reference to the domain file. The template is saved in HR's configuration directory.
- *Other* classes include exceptions for controlling file creation problems.

5.2.4 Testing Classes

A number of classes have been developed to support creating reformulations and performing tests of their performance.

ConstraintTestRunnerIter

The *ConstraintTestRunnerIter* class is responsible for controlling the overall testing process i.e. the loop of creating, running, assessing and saving test results. A TestRunner thread is spawned to run tests (allowing the GUI to be smoothly updated), see multithreading below.

ConTest

ConTest objects encapsulate an individual test scenario. This comprises the underlying solver formulation to be tested, the status of the test (tested/untested) and the results of the test when available. This class includes the *performTest* method which runs the test a given number of times using the testing predicate. Static data of the ConstraintTest class is used to initialise all tests on the same basis. This includes, for example, details of the test predicate that is used for testing and hence the data, or the predicate bindings, that each test must store when run.

Multithreading

Multithreaded test classes have been developed should we decide to do processing on multiple machines later. These processing threads interact with monitor queue classes for synchronized bringing of theorems from the theorem output file, creating and adding tests

- *TheoremBringer* Thread Accesses the HR output file and adds new theorems to the TheoremQueue.
- *TheoremQueue* maintains, and controls synchronized access to, a list of theorems from which test scenarios are to be generated.
- *TestFormer* accesses the TheoremQueue and uses the SolverFormulation method createTestFormulations to create tests to add to the TestQueue.
- *TestQueue* maintains, and controls synchronized access to, a list of tests to be run by TestRunner.
- *TestRunner* Runs any untested tests on the TestQueue.
- *PauseMonitor* To controll pausing and restarting. The overall process is controlled using boolean static variables of the thread classes to indicate the status of different threads.

This architecture will allow us to simultaneously run HR and test it's output as it arrives, rather than having to wait for HR to complete processing.

Utility Classes

A number of additional classes for file filters and useful functions have been developed.

5.2.5 Prolog Server Interface

The Prolog server is initialised by loading the a Prolog source file, which initialises a prologbeans server and registers a number of predicates for its use. This source file uses the prologbeans library, discussed in §3.3, which provides the functionality for starting a Sicstus server. The "main" goal is called, which registers a number of queries that can be called from the Java runtime, in much the same way as services are registered on a naming server when using Java RMI. The main goal then starts the server with the start/0 goal.

The nature of the approach means that the Prolog server would need to constantly change the declarations of predicates by re-consulting newly generated Prolog source files. Consequently, the Server Interface has been designed to be very simple but provide queries that allow files to be consulted and predicates registered at runtime, i.e. the capabilities of the server can be dynamically reconfigured. This flexibility means the application avoids having to constantly stop and restart a server with different capabilities.

5.2.6 Java Interface Classes

On the Java side, communication with Sicstus is handled through the SicstusInterface class. The class statically maintains a single SictusInterface instance, representing the applications connection to Sicstus. All interactions with the server are passed through this interface. This allows the other classes in the application to interact with Sicstus by simply calling static methods of the SicstusInterface class. This class has the following methods:-

- *launchSicstus/stopSicstus/connect* control launching the Sicstus executable and creating/destroying the Sicstus runtime.
- *getInterface* returns the SicstusInterface.
- *isRunning* reports whether a server is running.
- *setTimeout/getTimeout* control the time allowed for a query before a timeout exception is raised, (see §5.2.8.
- *runQuery/answer* runs a given query in the server and allows access to the answer returned. The prologbeans interface allows binding of values to predicate parameters prior to its being queried. However, it was found to be easier to create the predicate in whole as a String, by replacing parameters with values, prior to running the query. Supplementary exceptions classes have been developed to handle errors arising from running prolog queries, such as SicstusNotRunningException and SicstusQueryTimeoutException.
- *consult/resetConsults/registerQuery* allows new source files and predicates to be loaded and registered. These maintain a record of all consulted files and will not re-consult a file unless explicitly told to do so avoiding repetition. The list of consulted files can be cleared. A predicate in a consulted file must be registered to be usable.
- *setOutputFile/closeOutputFile/repointOutput/flushOutput* control re-direction of standard output to files on the local file system.

- *clearBindings/initBindings/getBindings* controls binding values to parameters prior to submitting a query.

5.2.7 Starting and Stopping Sicstus

The Sicstus instance is started using the java Runtime.getRuntime().exec method. The server is stopped by requesting the server run a halt/0 query.

5.2.8 Timeouts

The Prologbeans library provides a method to specify a timeout when a Java program sends a runQuery request to the Prolog server. After this timeout has completed, the server should stop running the query and the method return value should indicate timeout. However, the timeout function appears to have problems interrupting the Sicstus server, which becomes unresponsive following a timeout. Extensive remedial testing failed to solve the problem. A solution was developed using the Sicstus Prolog timeout library. This library provides the time_out/3 wrapper predicate, which seeks a given goal within a given time, unifying the third parameter with a success or timeout flag. This predicate does not add any noticeable overhead to the time taken for the goal and provides a simple and reliable way to control timeouts.

5.3 Development Environments

The application has been written in Java (1.4). GUI development was performed using the Netbeans IDE (www.netbeans.org). The ECLIPSE IDE (www.eclipse.org) was also used as it offers a number of additional useful features for pure coding. Prolog elements were developed using the Crimson text editor (www.crimsoneditor.com) and tested using the Sicstus Prolog application (www.sics.se/sicstus).

Chapter 6

Experiments

One of the main purposes of the system is to find ways to improve basic CSP specifications. It does this by interacting with the HR discovery system,(see §2.5), which provides theorems about the domain being investigated, translates those theorems into constraints (see chapter 4) and tests to see if those constraints improve the effectiveness of a basic CSP specification. We investigated a number of domains using the system. This chapter describes the set-up of a number of those experiments. In chapter 7 we describe the results of those experiments and, in chapter 8 we discuss and analyse those results.

6.0.1 Performing The Tests

The two major aspects of performing a reformulation experiment are running the HR process and translating, reformulating and testing the new formulation. Each test reformulation is generated into prolog code and the resulting file loaded into the prolog server. This code defines a predicate that will produce a solution to the problem. The effectiveness of this formulation is checked by running a user-defined test predicate, which will run that solver formulation as many times as are required to exhaust the search space (typically using the findall predicate). This test predicate returns statistics about the run.

6.0.2 Assessing the Results

In assessing the results and comparing different reformulations we have used CPU processing time as the measure of effectiveness. Measuring CPU time in a multithreaded processing environment is fine if the time reported is purely the time spent by the process on the CPU, excluding any switching time. The Sicstus Prolog manual states that this is the case but our testing showed there was often a disparity in times obtained when the computer was running under different loads, throwing doubt on this. There are other measures available to measure the performance of CSP solvers, such as *backtracks*, a count of the number of times the variable assignment process results in a dead end, or *resumptions*, which indicates the number of times when constraints were re-awoken. However, we could not easily identify a consistent relationship between the effectiveness of the specifications we were using and these measures either separately or together. Consequently, we have used the CPU time and performed multiple tests to reduce the chances of processor load adversely affecting the results.

6.0.3 Running HR

We did not perform any investigations into HR performance itself nor did we consider amending HR parameters, save for the number of steps. HR provided a vast number of conjectures, for example, in our study of Groups in HR produced over 1500 proven conjectures. When HR reaches a large number of conjectures we experience both a slowdown in performance of HR, as the overhead of comparing new implicates with old ones increases. In addition, there can also be a large increase in the memory footprint of HR. For example, in the case of Monoids, by the time HR had reached 5000 steps, its memory footprint was approximately 443MB. These factors mean that the length of an HR run has to be limited in some way. This may be a result of configuration settings of HR being incorrectly set or may be due to optimization issues with the code of HR. Should these issues be resolved then longer testing runs with more conjectures will be possible. HR was configured to search for prime implicates, as discussed in §2.5 although the output of HR used for testing was not restricted to prime implicates.

In all runs we only provided HR with single-size examples of the domain, which reduced the number of examples that HR had to begin with. This was done in the belief that HR could not accept multiple sizes of examples in its starting configuration which is not actually the case. HR can accept examples that vary in size in the same initial configurations files.

6.0.4 Domains of Investigation

Quasigroups were discussed in §2.8. In [2] quasigroup subtypes QG3 through QG7 were investigated, and we revisit them here. In addition, other subtypes of quasigroups, QG1 and QG2 have been investigated. A similar basic solver model was used for each sub-type of quasigroup, being that which produces Latin Squares of a given type. In order to create a basic model for the particular sub-type, the additional axiom for that sub-type was simply added in first order logic. This ease of set-up is noted in chapter 9, below. We have also transferred our investigations to other finite algebras namely Loops, Monoids, Semigroups and Groups.

6.0.5 Testing System

The following system was used for performing the tests:-

Computer	Toshiba Satellite P10-804
CPU	3.0 GHz Intel Pentium 4 (HT)
Memory	1,310MB
Operating System	Windows XP

Chapter 7

Results

This chapter provides details of the results we obtained for individual experiments using the ICaRuS system. We set up ICaRuS to investigate a number of domains and this section details the results obtained in running numerous batches of the system. We show further results in Section §7.13 where we analyse a selection of interesting theorems and investigate their effectiveness in improving CSP specification performance.

7.1 QG1 Quasigroups

These quasigroups have the additional axiom that:

$$\forall\, u\, v\, w\, x\, y\, z\ (x * y = u \wedge z * w = u \wedge v * y = x \wedge v * w = z) \rightarrow (x = z \wedge y = w)$$

which was added to the basic solver formulation for Latin Squares.

We ran HR for 50 minutes, in which time it produced 81 conjectures, using 4 examples of size 7 QG1 quasigroups. The test produced 24 reformulations of solver formulations that create examples of size 3 that improved on the base formulation by over 10%, featuring 18 different theorems. Note that one particular theorem

$$\forall\, b\, c\ \exists\, d\ (d * b = c)$$

appeared in every reformulation.

QG1 Results

No.	Theorems	Ave Time (ms)	% of base time
Base	-	817.7	100.0
1	*	692.7	84.7
2	$*\forall\, b\, c\ (\exists\, d\ (b * c = d))$	697.7	85.3
3	$*\forall\, b\, c\ (\exists\, d\, e\ (b * d = c \wedge e * c = d))$	703	86
4	$*\forall\, b\, c\ (\exists\, d\, e\ (b * b = d \wedge e * d = c))$	703	86
5	$*\forall\, b\, c\, d\ (\neg(b * d = c) \rightarrow \neg(b * d = c))$	708.3	86.6
6	$*\forall\, b\, c\ (\exists\, d\ (d * b = c) \rightarrow \exists\, e\ (e * b = b))$	708.3	86.6
7	$*\forall\, b\, c\ (\exists\, d\ (b * d = c)$	713.7	87.3
8	$*\forall\, b\, c\, d\ (\neg(b * d = c) \rightarrow \neg(b * d = c))$	713.7	87.3
9	$*\forall\, b\, c\, d\ ((b * c = d \wedge \neg(b * d = c)) \rightarrow \neg(b * d = c))$	718.7	87.9
10	$*\forall\, b\, c\, d\ ((b * c = d \wedge c * b = d \wedge \neg(b * d = c)) \rightarrow \neg(d * b = c))$	723.7	88.5
11	$*\forall\, b\, c\, d\ ((d * b = c \wedge \neg(b * d = c)) \rightarrow \neg(b * d = c))$	724	88.5
12	$*\forall\, b\, c\, d\ (\neg(b * d = c) \rightarrow\neq (b * d = c))$	729	89.2
13	$*\forall\, b\, c\, d\ ((b * c = d \wedge \neg(b * d = c)) \rightarrow \neg(b * d = c))$	729	89.2
14	$*\forall\, b\, c\, d\ ((b * d = c \wedge \neg(c * b = d)) \rightarrow \neg(c * b = d))$	729	89.2
15	$*\forall\, b\, c\, d\ (\neg(c * b = d) \rightarrow \neg(c * b = d))$	729.3	89.2
16	$*\forall\, b\, c\, d\ ((b * c = d \wedge d * b = c \wedge \neg(b * d = c)) \rightarrow \neg(d * c = b))$	729.3	89.2
17	$*\forall\, b\, c\, d\ (\neg(c * b = d) \rightarrow \neg(c * b = d))$	734.3	89.8
18	$*\forall\, b\, c\, d\ ((b * c = d \wedge \neg(c * b = d)) \rightarrow \neg(c * b = d))$	734.3	89.8
19	$*\forall\, b\, c\, d\ ((b * c = d \wedge \neg(b * d = c)) \rightarrow \neg(b * d = c))$	734.3	89.8
20	$*\forall\, b\, c\, d\ ((b * c = d \wedge d * b = c \wedge \neg(b * d = c)) \rightarrow \neg(b * d = c))$	734.3	89.8
21	$*\forall\, b\, c\, d\ ((b * c = d \wedge d * b = c \wedge c * d = b \wedge \neg(d * c = b)) \rightarrow \neg(b * d = c))$	734.3	89.8
22	$*\forall\, b\, c\, d\ ((b * c = d \wedge d * b = c \wedge \neg(d * c = b) \wedge \neg(b * d = c)) \rightarrow \neg(b * d = c))$	734.3	89.8
23	$*\forall\, b\, c\, d\ ((d * b = c \wedge \neg(d * c = b) \wedge \neg(b * d = c)) \rightarrow \neg(b * d = c))$	734.3	89.8
24	$*\forall\, b\, c\, d\ ((b * c = d \wedge b * d = c \wedge \neg(c * b = d)) \rightarrow \neg(c * b = d))$	734.7	89.8

***also used theorem** $\forall\, b\, c\ \exists\, d\ (d * b = c)$

The best reformulation was A, which used only $\forall\, b\, c\, \exists\, d\ (d*b = c)$ and found all solutions to the problem in 84.7% of the time taken by the base model.

7.2 QG2 Quasigroups

These quasigroups have the additional axiom that:

$$\forall\, u\, v\, w\, x\, y\, z\ (x * y = u \wedge z * w = u \wedge v * x = y \wedge v * z = w) \rightarrow (x = z \wedge y = w)$$

which was added to the basic solver formulation for Latin Squares. We ran HR for 2 hours 29 minutes, in which time it produced 77 conjectures. After one round of reformulations, 15 different theorems had been used in reformulations that beat the base formulation by over 10% in producing examples of size 3.

QG2 Results

No.	Theorems	Ave Time (ms)	% of base time
Base	-	1344.0	100.0
1	all b c (((exists d (b*c=d))))	679.5	50.6
2	all b c (((exists d (b*d=c))))	687.5	51.2
3	all b c (((exists d (d*b=c))))	703.0	52.3
4	all b c (((exists d (b*d=b))))	726.5	54.1
5	all b c d (((b*c=d & d*b=c & d*c=b & d*b=d) → (d*d=d)))	734.0	54.6
6	all b c d (((b*c=d & d*b=c & c*c=b & d*b=d) → (d*d=d)))	742.5	55.2
7	all b c d (((b*c=d & c*b=d & d*b=c & d*c=b) → (c*c=d)))	750.0	55.8
8	all b c d (((b*c=d & c*b=d & d*d=b & c*b=c) → (d*d=d)))	750.0	55.8
9	all b c d (((b*c=d & d*b=c & d*d=b & d*b=d) → (d*d=d)))	757.5	56.4
10	all b c d (((b*c=d & c*b=d & c*d=b & c*b=c) → (d*d=d)))	758.0	56.4
11	all b c d (((b*c=d & c*b=d & c*c=b & c*b=c) → (d*d=d)))	773.5	57.6
12	all b c d (((b*c=d & c*b=d & d*b=c & d*c=b) → (d*c=c)))	781.5	58.1
13	all b c d (((b*c=d & c*d=b & d*b=d & c*b=c) → (d*d=d)))	820.0	61.0
14	all b c (((exists d e (b*b=d & e*d=c))))	929.5	69.2
15	all b c (((exists d e (b*d=b & e*d=c))))	1016.0	75.6

The best performing reformulation used the theorem, $\forall\, b\, c\ \exists\, d\ (b * c = d)$ and took 50.6% of the time required by the base formulation to solve the given problem.

7.3 QG3 Quasigroups

These quasigroups have the additional axiom that:

$$\forall a\, b\ (a * b) * (b * a) = a$$

This was added to the basic solver formulation as the following conjecture.

all a b c d (((a*b=c) & (b*a=d)) → (c*d=a))

This axiom was added to the basic formulation for Latin Squares and 2 isomorphic examples were passed to HR for processing. HR ran for 25,000 steps taking 1 hour and 57 minutes, producing a total of 964 proven conjectures. In total, 12 reformulations outperformed the base formulation, in finding size 4 examples, by more than 10%, in terms of CPU time. These formulations used 10 of the theorems produced by HR in different combinations.

QG3 Results

Theorems	Ave Time (ms)	% of base time
Base	83.3	100.0
1,4	62.3	74.8
1,6	67.7	81.3
1,7	67.7	81.3
1	68	81.6
1,2	72.7	87.3
1,9	72.7	87.3
1,10	72.7	87.3
1,3	73	87.6
1,8	73	87.6
1,5	73.0	87.6
2,3	73	87.6
1,2	73.3	88.0

QG3 Theorems

No	Theorem	Appearances
1	all b c (((exists d (d*d=b))))	11
2	all b (((-(b*b=b)) → (-(b*b=b))))	3
3	all b ((-(b*b=b & -(b*b=b))))	2
4	all b c (((b*c=b & c*b=b) → (c*c=c)))	1
5	all b c (((b*c=c & c*b=c) → (c*c=c)))	1
6	all b c (((exists d (d*b=c))))	1
7	((-((exists b (b*b=b)))) → ((exists c (-(c*c=c)))))	1
8	all b (((b*b=b) → (-(b*b=b & -(b*b=b)))))	1
9	all b c (((-(c*c=c)) → (-(b*c=c & c*b=c))))	1
10	all b c ((-(b*b=c & -(b*b=c))))	1

The best performing model used a combination of theorems 1 and 4 and completed the solution search in 74.8% of the time taken by the base formulation.

7.4 QG4 Quasigroups

These quasigroups have the additional axiom that:

$$\forall\, a\, b\ (a * b) * (b * a) = b$$

This was added to the basic solver formulation as the following conjecture.

all a b c d (((a*b=c) & (b*a=d)) → (c*d=b))

This axiom was added to the basic formulation for Latin Squares and a single example was passed to HR for processing. HR ran for 10,000 steps which took a total of 2 hours and 3 minutes, producing a total of 242 proven conjectures. In total, 29 reformulations outperformed the base formulation by more than 10%, in terms of CPU time when finding examples of size 4. These formulations used 13 of the theorems produced by HR in different combinations.

QG4 Results

Theorems	Ave Time (ms)	% of base time
Base	99.0	100.0
1,2	62.3	62.9
1,3	68	68.7
1,7	72.7	73.4
1	73	73.7
1,4	73	73.7
1,9	73	73.7
1,5	78	78.8
1,6	78	78.8
1,11	78	78.8
1,12	78	78.8
8	83.3	84.1
2	83.3	84.1
1,8	83.3	84.1
1,13	83.3	84.1
1,10	83.7	84.5
1,2	88.3	89.2
3,4	88.3	89.2

QG4 Theorems		
No	Theorem	Appearances
1	all b c (((exists d (d*d=b))))	26
2	all b c (((b*c=b & c*b=c) → (b*b=c)))	4
3	all b c (((b*c=c & c*b=c) → (c*c=c)))	3
4	all b c (((b*c=b & c*b=c) → (c*c=c)))	3
5	all b c (((b*c=c & c*b=b) → (c*c=c)))	3
6	all b c (((b*c=b & b*c=c) → (b*b=c)))	3
7	all b c (((b*c=b & b*c=c) → (c*c=c)))	2
8	all b c (((exists d (b*d=c))))	2
9	all b c (((b*c=c & c*b=b) → (b*b=c)))	2
10	all b c (((b*c=c & -(c*c=c)) → (-(c*b=c))))	2
11	all b c (((b*b=c & -(c*b=b)) → (-(c*c=c))))	2
12	all b c (((b*b=c & -(b*c=b)) → (-(c*c=c))))	2
13	all b c d (((b*c=d & c*b=d) → (c*c=d)))	1

The best performing model used a combination of theorems 1 and 2 and exhausted the search space in 62.9% of the time required by the base model.

7.5 QG5 Quasigroups

These quasigroups have the additional axiom that:

$$\forall\, a\, b\ ((a * b) * a) * a = b$$

This was added to the basic solver formulation as the following axiom.

all a b c d (((b*a=c) & (c*b=d)) → (d*b=a))

HR found 88 proven conjectures for QG5 after running for 2,000 steps in 1 hour and 15 minutes. These were incorporated into the base formulation with the following results. A total of 5 reformulations outperformed the base formulation by 10% in terms of time. They made use of 5 different theorems in different combinations.

QG5 Results			
No.	Theorems	Ave Time (ms)	% of base time
Base	-	137.6	100.0
1	all b c (((b*c=c) → (c*b=c)))	115.6	84.0
2	all b c (((b*b=c & b*c=b) → (b*b=b)))	121.8	88.5
3	all b c (((b*c=c & c*b=b) → (b*b=b)))	121.8	88.5

The best performing reformulation made use of the theorem $\forall\, b\, c\ (b * c = c) \rightarrow (c * b = c)$ and exhausted the search space in 84.0% of the time required by the base formulation.

7.6 QG6 Quasigroups

These quasigroups have the additional axiom that:

$$\forall a\, b\ ((a * b) * b = a * (a * b))$$

This was added to the basic solver formulation in the following way:

all a b c d (((a*b=c) & (c*b=d)) → (a*c=d))

HR found 40 proven conjectures for QG6 after running for 10,000 steps in 26 minutes using 1 example. These were incorporated into the base formulation and, in the first round of reformulations, a total of 7 reformulations outperformed the base formulation by 10% in terms of time, when searching for examples of size 4.

QG6 Results

No.	Theorems	Ave Time (ms)	% of base time
Base	-	83.3	100.0
1	all b c ((b*b=b))	41.7	50.1
2	all b c ((((exists d (d*b=c))) → (b*b=b)))	57.3	68.8
3	all b c (((b*b=c) → (b*c=b)))	62.3	74.8
4	all b c (((b*b=c) → (b*c=c)))	62.3	74.8
5	all b c (((exists d (b*d=c))))	67.7	81.3
6	all b c (((exists d (b*c=d))))	67.7	81.3
7	all b c (((exists d (d*b=b))))	67.7	81.3

Only the results of the first round of reformulations are shown. Further rounds of reformulation simply used combinations of these axioms. The best formulation found by the system used a combination of theorem 1 together with the other theorems in the table, with the best performing formulation exhausting the search space in approximately 31 ms.

7.7 QG7 Quasigroups

These quasigroups have the additional axiom that:

$$\forall a\, b\ ((b * a) * b = a * (b * a))$$

This was added to the basic solver formulation in the following way:

all a b c d (((b*a=c) & (c*b=d)) → (a*c=d))

HR provided 122 proven conjectures after 32 minutes of processing. When these conjectures were tested with the base formulation and generated a total of 5,374 different reformulations using these conjectures. After the first round of testing there were 12 formulations that outperformed the base formulations by more than 10%.

QG7 Results

No.	Theorems	Ave Time (ms)	% of base time
Base	-	583.3	100.0
1	all b c ((b*b=b))	203.0*	34.9
2	all b c ((((exists d (d*b=c))) → (b*b=b)))	203.3	34.9
3	all b c (((exists d (d*b=b & b*b=d))))	208.3	35.7
4	all b c (((b*c=c) → (b*b=c)))	239.7	41.1
5	all b c (((b*b=c) → (c*b=b)))	323.0	55.4
6	all b c (((c*b=b) → (b*b=c)))	369.7	63.4
7	all b c (((b*b=c) → (b*c=c)))	401.3	68.8
8	all b c (((b*c=b) → (b*b=c)))	411.7	70.6
9	all b c (((b*b=c) → (c*b=c)))	447.7	76.8
10	all b c (((b*b=c) → (b*c=b)))	474.0	81.3
11	all b c (((c*b=c) → (b*b=c)))	479.3	82.2
12	all b c (((exists d (b*c=d))))	510.3	87.5

***see below, regarding further reformulations**

Note we have only included the results of the first round of reformulations for clarity. In the full test we had a very large number of combinations of the above theorems in different reformulations with the most effective formulations achieving speeds of approximately 109ms which represents 18.7% of the time required by the base model.

7.8 Loops

Loops are quasigroups with the additional axiom that:

$$\exists e \, \forall a \, (e * a = a * e = a)$$

which was added to the basic solver formulation for Latin Squares to get a basic solver formulation for loops. We ran HR for 26 minutes covering 5,000 steps. HR produced 126 conjectures. Upon testing, however, we did not identify any useful theorems that could be added to the basic loop formulation to achieve a significant speed boost.

7.9 Semigroups

The basic formulation for a Semigroup is obtained from the basic multiplication table solver formulation. This is simply the starting point for quasigroups with the all_different constraints removed, which are interpretations of the quasigroup divisor axioms. To this multiplication table formulation, the axiom of associativity is added.

all a b c ((a * b) * c = a * (b * c))

This was translated into the following first order logic statement for use in the basic formulation

all a b c d e ((a*b=d & b*c=e) → (d*c = a*e))

The basic formulation produced 22 examples of semigroups with which we ran HR for 43 minutes, producing 113 conjectures after 10,000 steps. No formulations were produced that significantly improved the results of the basic specification. The best performing reformulations, for finding size 3 examples, are shown here.

Semigroup Results

No.	Theorems	Ave Time (ms)	% of base time
Base	-	177.3	100.0
1	all b c (((b*c=b & b*c=c) → (b*b=c)))	167.0	94.2
2	all b c (((b*c=b & b*c=c) → (b*b=b)))	177.0	99.8
3	all b ((((exists c (c*b=b))) → ((exists d (b*b=d)))))	177.0	99.8

The best performing reformulation made use of the theorem $all\,b\,c\ (b * c = b \wedge b * c = c) \rightarrow (b * b = c)$ and exhausted the search space in 94.2% of the time required for the base formulation.

7.10 Monoids

Monoids are semigroups with an identity element. Consequently, the starting formulation for Monoids was created by simply adding the identity element axiom to the basic formulation for semigroups.

exists a (all b ((a * b = b) & (b * a = b)))

The basic formulation provided 7 examples of monoids to HR. We ran HR for 10,000 steps over 37 minutes generating 204 proven conjectures.

Monoid Results

No.	Theorems	Ave Time (ms)	% of base time
Base	-	114.3	100.0
1	all b c ((((exists d (b*d=c))) → ((exists e (b*e=b)))))	104.3	91.3
2	all b c (((b*c=b & b*c=c) → (b*b=c)))	109.3	95.6

The best reformulation used theorem $\forall\, b\, c\ (\exists\, d\ (b * d = c) \rightarrow \exists\, e\ (b * e = b))$ and exhausted the search space, for size 3 examples, in 91.3% of the time taken by the base formulation.

7.11 Groups

Groups are monoids where every element of the set has a left and right inverse element. Consequently, the starting formulation for Groups was created by adding this axiom to the basic formulation for Monoids. A little modification was made, namely to select one of the elements as the identity, which is acceptable as any solutions that this removes would be isomorphic to at least one of those still found. The identity axiom was amended to

all a (1 * a = a & a * 1 = a)

And the axiom for inverses was added as

all a (exists b (a * b = 1 & b * a = 1))

We ran HR for 10,000 steps over 5 hours and 15 minutes, in which time it generated 1,542 proven conjectures on Groups, from the 2 non-isomorphic examples and starting concepts. Many reformulations were found that improved significantly upon the basic specification. We show only those for which the improvement over the base specification was more than 40% when searching for examples of size 3.

Group Results

No.	Theorems	Ave Time (ms)	% of base time
Base	-	115.0	100.0
1	all b c (((exists d e (c*d=b & e*c=d))))	31.0	27.0
2	all b c (((exists d e (d*c=b & e*d=b))))	52.0	45.2
3	all b c (((exists d (b*d=c))))	57.3	49.8
4	all b c (((exists d (d*b=c))))	62.3	54.2
5	all b c ((((exists d (b*c=d & d*d=d))) → ((exists e (b*e=c & c*c=e)))))	67.7	58.9
6	all b c (((-(b*b=c)) → ((exists d (b*d=c & -(d*d=c))))))	67.7	58.9
7	all b c ((((exists d (b*d=c & d*d=c & -(b*c=d)))) → (-(c*c=c))))	67.7	58.9
8	all b c ((((exists d (b*c=d & d*d=d))) → ((exists e (e*b=c & c*c=e)))))	68.0	59.1

The best performing reformulation used the theorem $\forall b\,c\ (\exists d\,e\ (c * d = b \wedge e * c = d))$ and exhausted the search space in 27.0% of the time required by the base formulation.

7.12 Graphical Summary of Results

We present in figure 7.1 a graph which shows, for each sub-type of quasigroup the improvement the best reformulation represented, in terms of speed-reduction, against the base model. For consistency, we show for all sub-types the percentage of base model time rather than absolute speed.

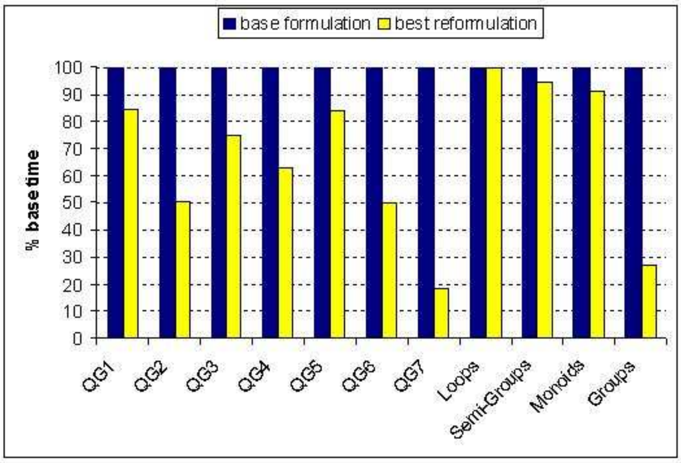


Figure 7.1: Results Summary

7.13 Further Reformulation Results

Some interesting theorems were found during the ICaRuS test running. We have performed a brief supplementary review of some of the quasigroup theorems them to consider their impact upon reformulation performance when tested with domains of different sizes.

7.13.1 QG1 Quasigroups

We performed one reformulation for QG1 quasigroups using the additional theorem identified by the ICaRuS run and did some very simple testing. The table below shows the results of running the base case and the reformulation in a search for quasigroups of sizes 3 and 4 using lexicographical and fail-first search heuristics (see §2.2).

		B	R_1
Size	time(ms)	782	734
3	%	100.0	93.8
Size	time(ms)	42,960	17,906
4	%	100.0	41.6

R_1 - $\forall\, b\, c\ \exists\, d(d * b = c)$

7.13.2 QG2 Quasigroups

We performed three reformulations for QG2 quasigroups using examples of the subsets of additional theorems identified by the ICaRuS run and did some very simple testing.

		B	R_1	R_2	R_3
Size	time(ms)	1,344	734	750	781
3	%	100.0	54.6	55.8	58.1
Size	time(ms)	67,719	16,890	46,922	47,204
4	%	100.0	24.9	69.3	69.7

R_1 - $\forall\, b\, c\ \exists\, d(d * b = c)$
R_2 - $\forall\, b\, c\, d\ (b * c = d \& c * b = d \& c * c = b \& c * b = c) \rightarrow (d * d = d)$
R_3 - $\forall\, b\, c\, d\ (b * c = d \& c * b = d \& d * b = c \& d * c = b) \rightarrow (d * c = c)))$

7.13.3 QG3 Quasigroups

We re-tested the interesting theorems found in the bulk testing. Theorem 4 did not prove to be of significant value when scaled to higher order domains so we only consider reformulations using theorem 1, $\forall\, b\ \exists\, d\ (d * d = b)$, which are denoted R_1. In addition, we consider reformulations using the all_different on the diagonal constraint as reformulation R_2.

		B	R_1	R_2
Size 4	time(ms) %	78.1 100.0	71.9 92.1	64.1 82.1
Size 5	time(ms) %	968.8 100.0	829.7 85.6	684.4 70.6
Size 6	time(ms) %	23,453.1 100.0	22,017.4 93.9	13,735 58.6

These results are shown graphically in figure 7.2.

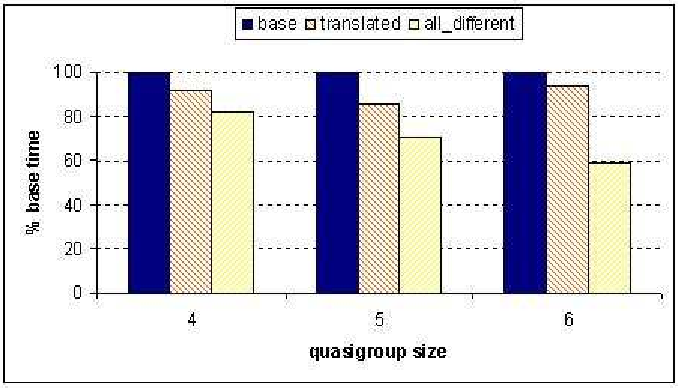


Figure 7.2: QG3 Results by Size

7.13.4 QG5 Quasigroups

We performed one reformulation for QG5 quasigroups using the additional theorem identified by the ICaRuS run and did some very simple testing. The other theorems did not significantly benefit the performance of our solvers.

		B	R_1
Size 4	time(ms) %	1,276 100.0	1,005 78.7
Size 5	time(ms) %	21,390 100.0	15,250 71.3

R_1 - $\forall b\, c\ (b * c = c \rightarrow c * b = c)$

7.13.5 QG6 Quasigroups

We investigated three reformulations for QG6 quasigroups using two of the theorems identified in the ICaRuS run.

		B	R_1	R_2
Size	time(ms)	83.3	36.3	62.7
4	%	100.0	43.6	75.3
Size	time(ms)	401.0	130.3	568.0
5	%	100.0	32.5	141.6
Size	time(ms)	3,093.7	593.7	9,354.3
6	%	100.0	19.2	302.4

R_1 - $\forall\, b\, c\ (b * b = b)$
R_2 - $\forall\, b\, c\ (\exists d(d * b = c)) \rightarrow (b * b = b)$

7.13.6 QG7 Quasigroups

We investigated three reformulations for QG7 quasigroups using two of the theorems identified in the ICaRuS run.

		B	R_1	R_2
Size	time(ms)	593	203	250
5	%	100.0	34.2	42.2
Size	time(ms)	6875	1093	2688
6	%	100.0	15.9	39.1

R_1 - $\forall\, b\ (b * b = b)$
R_2 - $\forall\, b\, c\ (b * c = c) - > (b * b = c)$

Chapter 8

Analysis of Results

In this chapter we analyse the results shown in §7 and give some commentary about the theorems found there and consider why they were found to be useful by the system.

8.1 QG1 Quasigroups

$\forall\ b\ c\ \exists\ d(d{*}b = c)$is the only real theorem of note as it appears in every improving reformulation and, when featuring alone, performed the best in all testing. The axiom is one of the conjuncts of the quasigroup axioms and consequently, should already be included as a constraint in the basic formulation. However, this added constraint gives some improvement, possibly because it is a slightly different formulation of those axioms, which have been translated as all_different constraints.

8.2 QG2 Quasigroups

Theorems 1-4,14 and 15 are, essentially, restatements of the axioms of quasigroups and have been discussed with earlier results.

Theorems 5,6,8,9,10 and 11 are interesting.

$$\forall\ b\ c\ d\ (b*c = d \wedge d*b = c \wedge d*c = b \wedge d*b = d) \rightarrow (d*d = d)$$
$$\forall\ b\ c\ d\ (b*c = d \wedge d*b = c \wedge c*c = b \wedge d*b = d) \rightarrow (d*d = d)$$
$$\forall\ b\ c\ d\ (b*c = d \wedge c*b = d \wedge d*d = b \wedge c*b = c) \rightarrow (d*d = d)$$
$$\forall\ b\ c\ d\ (b*c = d \wedge d*b = c \wedge d*d = b \wedge d*b = d) \rightarrow (d*d = d)$$
$$\forall\ b\ c\ d\ (b*c = d \wedge c*b = d \wedge c*d = b \wedge c*b = c) \rightarrow (d*d = d)$$
$$\forall\ b\ c\ d\ (b*c = d \wedge c*b = d \wedge c*c = b \wedge c*b = c) \rightarrow (d*d = d)$$

In all cases, the antecedent is only true when c=d. Therefore, taking the first as an example, it reduces as follows to:

$$\forall\ b\ c\ (b * c = c \wedge c * b = c \wedge c * c = b \wedge c * b = c) \rightarrow (c * c = c)$$
$$\forall\ b\ c\ (b * c = c \wedge c * b = c \wedge c * c = b) \rightarrow (c * c = c)$$

In essence, these constraints trigger failure under certain conditions involving three other variables. Given the complexity of the basic axiom of QG2 quasigroups it is unsurprising that such structures give a performance benefit.

Theorems 7,12 and 13 are interesting also:

$$\forall\ b\ c\ d\ (b * c = d \wedge c * b = d \wedge d * b = c \wedge d * c = b) \rightarrow (c * c = d)))$$
$$\forall\ b\ c\ d\ (b * c = d \wedge c * b = d \wedge d * b = c \wedge d * c = b) \rightarrow (d * c = c)))$$
$$\forall\ b\ c\ d\ (b * c = d \wedge c * d = b \wedge d * b = d \wedge c * b = c) \rightarrow (d * d = d)))$$

These, again, trigger failure when the four variable values in the antecedent are assigned values in a particular way. The consequent results in either an existing variable assignment or an all_different constraint being violated, which means that any assignment of a,b,c and d that made the antecedent would be impossible for QG2 quasigroups.

8.3 QG3 Quasigroups

The best performing reformulation included both theorems 1 and 4. In the automated testing, this reformulation was approximately 25.2% faster at exhausting the search space than the base formulation. Its performance is reviewed further, below.

Theorem 1, $\forall\ b\ \exists\ d\ (d*d = b)$, in mathematical terms, states that every element is the result of at least one multiplication of some other element by itself. In structural terms this means that each element must appear at least once on the diagonal of the Latin Square from top left to bottom right. This theorem was also identified in [2]. The fact that every element appears on the diagonal and the size of the diagonal is equal to the domain size, this means that each of the elements on the main diagonal must be distinct from each of the others. Consequently, in [2] this was interpreted as an all_different constraint on the variables along the diagonal.

The all_different constraint is a very powerful constraint as it allows for the domains of multiple variables to be adjusted when a variable instantiation is tried. The reformulation in [2] using the all_different constraint took approximately 56% as long as their base formulation to exhaust the search space, which is a much bigger increase than the improvement we saw in our results. This is because the all_different constraint is a single constraint that propagates to all other variable domains at the same time. The translation of this theorem by our system will have resulted in a number of individual constraints which will take more time to propagate. The translation performed by this system to create the constraints does not go as far as to identify this as an all_different constraint, and hence the improvement in performance is not as stark. However, this did raise the interesting question of whether such a logical deduction could be made and this is discussed in chapter 9.

Theorem 4, $\forall\ b\ c\ (b{*}c = b \wedge c{*}b = b) \rightarrow c{*}c = c$, provides an improvement. This implication allows the solver to quickly instantiate the value of c*c whenever it has assigned the same value to commuting pairs. However QG3 quasigroups are anti-abelian i.e. $\forall\ a\ b\ (a{*}b = b{*}a \rightarrow b = a)$, therefore the improvement in solver speed must result from the fact that instantiating c*c to c leads, more quickly, to failure. In the testing, this gave a marginal improvement over theorem 1 alone.

This application of the anti-abelian law is different to the theorem that was identified in [2]. They identified $\forall\ a\ b\ \exists\ c(a * c = b \wedge c * a = b) \rightarrow a * a = b$ as being potentially interesting. They used this axiom, and the basic quasigroup axioms to mathematically derive the fact that QG3 quasigroups are Anti-Abelian. In structural terms this means that the elements of a Latin Square $\forall iji{\neq}j(X_{ij}{\neq}X_{ji})$ where X_{nm} is on the n^{th} row and m^{th} column. This represents a set of very strong constraints on the elements opposite each other across the central diagonal, and means that the solver can, when one has been assigned, remove it's value from the domain of the other variable. Consequently, this also resulted in significant efficiency gains over their base formulation. In fact, in terms of speed, when smallest domain ordering was applied, this constraint improved their basic formulation by nearly 98%. Although our theorem is simpler, it does not even approach the performance of their reformulation using simple constraints. Again, the additional step of logic from implication conjecture to a set of difference constraints was not performed by our system, this is discussed in chapter 9.

Theorem 5 $\forall\ b\ c\ (b * c = b \wedge c * b = b) \rightarrow c * c = c$ is a slight reformulation of theorem 4. However, it did not add any effectiveness to theorem 1 alone in testing.

Theorem 6, $\forall\ b\ c\ \exists\ d(d * b = c)$, states that, for every pair of elements b and c, c must appear as something * b i.e. c must be somewhere in column b, which is one of the conjuncts of the quasigroup axioms, $\forall\ a\ b\ \exists\ x\ y\ (a * x = b \wedge y * a = b)$. However, because the quasigroup axioms have been translated into all_different constraints, this theorem seems to give a different guide to the search with a slight improvement in results.

A number of axioms that appear in the results table are quite trivial. For example, theorem numbers 2, 3 and 10 are tautologies and in theorem 8, the antecedent is irrelevant as the consequent is a tautology. Alone, they do not lead to improvements in the solver efficiency.

Note that no theorem containing more than 4 variables produced a reformulation that improved upon the base formulation. In addition, the theorem $\forall\ a\ b\ (a * a = b \rightarrow b * b = a)$, which was discovered by HR in [2] was not discovered in the testing we performed. This is possibly due to the search settings of HR.

When looking at the performance of constraints in the search for larger order examples, we compared the performance of the constraints to the all_different constraint which could be used in its place. The all_different constraint performed much better, leading us to suggest that some kind of process that could make the translation from first order constraints to all_different constraints could be very beneficial.

8.4 QG4 Quasigroups

The best performing reformulation included both theorems 1, $\forall\ b\ c\ \exists\ d\ (d * d = b)$, and 2. In the automated testing, this reformulation was approximately 37% faster at exhausting the search space than the base formulation.

Again, the most useful theorem is $\forall\ b\ \exists\ d\ (d * d = b)$, which is discussed in the previous section.

Theorems 3 and 13 are both variants of the two theorems, 4 and 5, seen in the QG3 results analysis of §8.3. They exploit the fact that, like QG3 quasigroups, QG4 quasigroups are anti-abelian, using this fact in different ways to control the search of the solution space.

Theorems 2-7 and 9 are all very interesting and all involve the same basic property of QG4 quasigroups.

$$\begin{array}{l}
\forall\ b\ c\ (b * c = b \wedge c * b = c) \rightarrow (b * b = c) \\
\forall\ b\ c\ (b * c = b \wedge c * b = c) \rightarrow (c * c = c) \\
\forall\ b\ c\ (b * c = c \wedge c * b = b) \rightarrow (c * c = c) \\
\forall\ b\ c\ (b * c = b \wedge b * c = c) \rightarrow (b * b = c) \\
\forall\ b\ c\ (b * c = b \wedge b * c = c) \rightarrow (c * c = c) \\
\forall\ b\ c\ (b * c = c \wedge c * b = b) \rightarrow (b * b = c)
\end{array}$$

They state that for pairs of elements X_{ij} and X_{ji}, i.e. opposite each other over the main diagonal, if one of them is assigned i then the other should not be assigned j and vice versa. This is enforced by the solver by the above implicates because, whenever such an assignment takes place, this constraint forces the solver to assign one of those values to the main diagonal in either the same column or row as an existing assignment of that same value, breaking the all_different constraint. This restriction can be stated mathematically as $\forall\ b\ c\ (b * c = b \rightarrow c * b{\neq}c)$ and $\forall\ b\ c\ (b * c = c \rightarrow c * b{\neq}b)$ whenever $b \neq c$. A sample proof is shown in figure 8.1.

Theorems 10,11 and 12 are re-writings of previous theorems, they can be derived as shown in figure 8.2. They therefore affect the performance of the solver

1	$\forall\ a\ b\ (a * b) * (b * a) = b$	qg4 Axiom
2	$b * c = b$	given
3	$b \neq c$	given
4	$c * b = c$	assume
5	$(b * c) * (c * b) = c$	1
6	$b * (c * b) = c$	2,5
7	$b * c = c$	6,4
8	$b = c$	2,7
9	$c * b \neq c$	4,3,8

Figure 8.1: QG4 sample proof

$$\begin{gathered}A \wedge B \rightarrow C \\ \neg(A \wedge B) \vee C \\ \neg A \vee \neg B \vee C \\ \neg A \vee C \vee \neg B \\ \neg A \vee \neg\neg C \vee \neg B \\ \neg(A \wedge \neg C) \vee \neg B \\ (A \wedge \neg C) \rightarrow \neg B\end{gathered}$$

Figure 8.2: QG4 Derivation of Theorems 10,11 & 12

agent for similar reasons to those theorems.

Theorem 8 is analogous to theorem 6 in QG3 stating that each value should appear in each row.

Note that as for QG4, no theorem containing more than 4 variables produced a reformulation that improved upon the base formulation. In addition, the theorem $\forall\ a\ b\ (a * a = b \rightarrow b * b = a)$, which was discovered by HR in [2] was not discovered in the testing we performed. This is possibly due to the search settings of HR. We tested this theorem separately, however, and the results suggested that our system would not have identified it as useful.

8.5 QG5 Quasigroups

Theorem 1, $\forall\ b\ c\ (b * c = c \rightarrow c * b = c)$ provides some guidance to the solver whenever it places, in row b, the value equal to the column it is placing, i.e. c, it immediately knows to place the same value in the element opposite across the diagonal. This theorem was discovered in [2].

Theorem 2, $\forall\ b\ c\ (b * b = c \wedge b * c = b) \rightarrow b * b = b$ is very interesting. It tells the solver that if it ever places the value c in row b on the central diagonal then it should *not* place the value b in column c of that same row, otherwise this implication tells it to try to also place b in the central diagonal, causing a failure because it is already assigned to c. It is derived mathematically in figure 8.5.

1	$\forall\ a\ b\ ((a * b) * a) * a = b$	qg5 Axiom
2	$\forall\ a\ b\ (a * b = b * a)$	abelian
3	$b * b = c$	given
4	$b * c = b$	given
5	$((b * c) * b) * b) = c$	1,2
6	$(b * b) * b = c$	5,4
7	$c * b = c$	6,3
8	$c * b = b$	2,4
9	$c = b$	7,8
10	$b * b = b$	9,3

Theorem 3, $\forall\ b\ c\ (b * c = c \wedge c * b = b) \rightarrow (b * b = b)$ is a variation of theorem 1. It breaks the all_different constraint on row b whenever b and c are unequal and the antecedent holds, i.e. whenever theorem 1 is not applied. Placing b in b*b will cause a backtrack because c*b is already assigned the value b.

8.6 QG6 Quasigroups

Theorem 1, $\forall\ a(a * a = a)$ is the well-known idempotent property of QG6 quasigroups and, as such, would normally be included as part of the axioms of the domain. However, by not including this property, we have discovered other interesting conjectures. Theorem 2 is another variant on this conjecture $\forall\ b\ c\ (\exists d(d{*}b = c)) \rightarrow (b{*}b = b)$, which would trigger the assertion of the idempotency property under certain, guaranteed, circumstances during the search.

Theorems 3 $\forall\ b\ c\ (b * b = c) \rightarrow (b * c = b)$, is not easy to interpret in terms of why it improves the basic formulation. It constrains the solver so that, if it places anything other than b in b*b then it must place a b on the same row in column c, this must propagate to a failure more quickly.

Theorem 4,$\forall\ b\ c\ (b * b = c) \rightarrow (b * c = c)$, enforces the idempotency rules in an indirect manner. It constrains the solver to place the value c in two places on the same row if c is anything other than b. Therefore, breaking idempotency will lead the solver to break an all_different constraint.

Theorems 5,6,7 have all been discussed in earlier results sections.

Overall, the only theorems found to be useful made use, either directly or indirectly, of the idempotency property. If this was stated as an axiom then the utility of all the other stated theorems would vanish.

8.7 QG7 Quasigroups

Theorem 1, $\forall\ a(a * a = a)$ appears again, and is a very powerful constraint, QG7 quasigroups are all idempotent. Theorems 2 and 3 indirectly also enforce this this.

Theorems 4,6,8 and 11 are direct results of the idempotency property. They work well as constraints because they place direct and simple restrictions on the values that can be assigned to variables.

$$\begin{array}{c} \forall\ b\ c\ (b * c = c) \rightarrow (b * b = c) \\ \forall\ b\ c\ (c * b = b) \rightarrow (b * b = c) \\ \forall\ b\ c\ (b * c = b) \rightarrow (b * b = c) \\ \forall\ b\ c\ (c * b = c) \rightarrow (b * b = c) \end{array}$$

They enforce additional restrictions on the values in the elements off the main diagonal. In effect they state that a multiple can never be equal to either of the multiplier or the multiplicand, that is reserved for the central diagonal.

$$\forall\ a\ b\ a \neq b \rightarrow (a * b \neq a \wedge a * b \neq b)$$

As soon as the solver has assigned one of these values incorrectly then, by following this constraint, it assigns an incorrect value to the main diagonal which eventually breaks the assignments. They work well as constraints but they are superfluous if the idempotency constraint is also posted.

Theorems 5,7,9 and 10 are similar to those above and, in a sense, enforce the idempotency rule by triggering a failure of one of theorems 4,6,8 or 11.

$$\begin{array}{c} \forall\ b\ c\ (b * b = c) \rightarrow (c * b = b) \\ \forall\ b\ c\ (b * b = c) \rightarrow (b * c = c) \\ \forall\ b\ c\ (b * b = c) \rightarrow (c * b = c) \\ \forall\ b\ c\ (b * b = c) \rightarrow (b * c = b) \end{array}$$

If the solver assigns a value to the central diagonal then it is forced, by this constraint, to assign a value to a multiple that is either the multiplier or the multiplicand. These are useful constraints because they directly link the value of pairs of variables and this can lead to quicker backtracks.

Theorem 12 has been covered in earlier results discussions.

8.8 Loops

No improving reformulations were found. In some ways, this is unsurprising. The values of the variables in Loops are restricted in only a very minor way. That is, for one column and one row representing the identity element, the variables will follow the labelling of the row and column numbers. All other variables can be assigned by following just the Latin Square property, which leaves plenty of options. This means they, perhaps, lack a lot of the structure that leads to the kind of useful constraints that we have seen earlier.

8.9 Semigroups

The results for semigroups did not identify a particular theorem that improved the base formulation. The two theorems that feature in the results table

$$\forall\ b\ c\ (b * c = b \& b * c = c) \rightarrow (b * b = c)$$
$$\forall\ b\ c\ (b * c = b \& b * c = c) \rightarrow (b * b = b)$$

are superfluous - they can only be triggered when b=c and then have no effect. Hence they do not improve efficiency. These theorems were, therefore, not tested further.

8.10 Monoids

Similarly to semigroups, the results for Monoids failed to clearly identify any beneficial theorems. The first of the semi-group theorems appeared, as did

$$\forall\ b\ c\ \exists\ d\ (b * d = c) \rightarrow \exists\ e\ (b * e = b)$$

but they did not give significant performance improvements and it is difficult to see, in any case, why they would.

8.11 Groups

The results for Groups showed a large number of theorems which improved the basic formulation.

The first four theorems can be considered, by splitting the conjunctions, as the following set of statements, the superscript indicates which theorem they came from.

$$\forall\ b\ c\ \exists\ d\ (c * d = b)^1$$
$$\forall\ b\ c\ \exists\ d\ (d * c = b)^2$$
$$\forall\ b\ c\ \exists\ d\ (b * d = c)^3$$
$$\forall\ b\ c\ \exists\ d\ (d * b = c)^4$$
$$\forall\ b\ c\ \exists\ d\ e\ (e * c = d)^1$$
$$\forall\ b\ c\ \exists\ d\ e\ (e * d = b)^2$$

They each place very strong restrictions on the variables. For example, the first theorem states that every value is the result of multiplying every other value by some value. In mathematical terms, they are stating that groups are quasigroups, which is true for all groups and follows from the inverses and associativity axiom. Structurally, this means that all values appear in all rows and all columns, as per quasigroups.

$\forall\ b\ c\ \exists\ d\ (b * c = d \wedge d * d = d) \rightarrow (\exists\ e\ (b * e = c \wedge c * c = e))$ is interesting. In groups, d*d=d can only be true if d is the identity element. It is not true for any other elements. Therefor, in the above antecedent b and c are inverses. This means the consequent states that the column number where c appears in row b should be assigned to c*c.

There are many other group results, many of which, prima facie, appear to express similar relationships between groups of elements. This suggests a large and interesting number of complex interactions between the variables. However, a full study of all the group results has been left for further work.

Chapter 9

Conclusions

In this chapter we discuss the conclusions we have drawn from our work in developing the system and from performing the experiments discussed in §8.

9.1 Success of the Approach

The system successfully identified a number of interesting theorems. Amongst others, it successfully identified most of the theorems, or equivalent theorems, that were identified in [2]. The system was successful in identifying interesting mathematical properties of the domains being investigated from within the large volume of data produced by HR. This, to a great extent, validates the approach taken and shows that such automation is feasible in practice, which encompasses the translation of theorems, the automation of testing HR's output and the assessment and reporting of the results. In addition, we saw that it was possible to use the system to formulate CSPs, review the output from HR and create improving reformulations without the user having expert skills in either constraint solving or the domain of investigation.

9.2 Creating Basic Formulations

It was possible to create basic formulations for each of the algebras under investigation by simply adding axioms in first order logic. This significantly reduced the set-up time for performing the experiments. Indeed, trying to formulate a base formulation in the language of the solver proved to be a very skilled, extremely difficult and time-consuming task. However, the translation functions of the system could be used in place of such expert knowledge and we significantly reduced the effort required here. In addition to creating basic formulations, the system proved very useful in cutting down the HR configuration time. Both these results represent an improvement on the original method.

9.3 all_different

In some cases, for example the theorem $\forall\ a\ \exists\ b\ (b * b = a)$, it is possible to translate the theorem into one, or possibly more, sets of all_different constraints. We showed in our review of QG3 that this translation, if achieved, would significantly improve the performance of a reformulation. We have performed additional investigations into this. We have looked at combining our translated theorems with conjectures about the equality or otherwise of different variables. We can then ask the solver to find examples that satisfy the combination. The solver's success or failure allows us to reason about the relative values of different variables in light of a given theorem. We suggest further work to investigate the feasibility of this in chapter 10.

9.3.1 Conjecture Size And Complexity

We noted, in our results for QG3 quasigroups, that no theorem involving more than three variables led to a speed up in a reformulation. It would appear that the complexity of a conjecture bears a direct relationship with its usefulness as a constraint, whereby, in general, the fewer variables and operators in a theorem, the better the chance it performs well when translated into a constraint. For instance, the axiom:

all a (exists b (b * b = a))

which was found to be true for both QG3 and QG4 performed best. It may well be that there is a balance between the cost to the solver of propagating complex constraints with the benefit of constraining the variables. It may also be the case that this is a result of the translation approach. Further work would be required to investigate this.

9.3.2 Optimization

In almost all cases, we used the axiom translation functions to generate the basic starting formulations. An expert in the field of constraints programming would probably be able to produce much better basic formulations that outperform those we have used and it is also possible that these expert basic formulations would outperform the best reformulations that our process uncovered, leading to suggestions that the process is inherently flawed. However, the process is not designed with the sole aim of finding the best possible reformulation for solving a given problem, although this would be a long term goal. Rather, the system has a number of purposes such as assessing potentially useful HR output. We believe the results of such a review should be of equal interest to an expert constraints programmer, who would be able to take the theorems indicated as useful by the system and convert them into highly optimized new constraints.

9.4 Overall Conclusion

We have investigated an automated approach to applying machine learning and theorem proving to the problem of formulating constraint satisfaction problems. We began with the hypothesis, in chapter 1, that it was possible to fully automate the method of constraint reformulation described in [2] and that the level of expertise required to perform investigations of this nature could be significantly reduced. We believe the results above show that this hypothesis is true.

We built an application to allow users to perform studies of this nature. This application provides the user with everything they need to set up, run, monitor and assess these studies as well as functions to allow them to generate CSP formulations, translate theorems into constraints and perform other ad-hoc tasks. Using this application, we have investigated a number of domains within finite algebra and shown how our automated method facilitates easy transfer between domains.

We believe the study has been a success and we have even discovered some interesting new properties of Quasigroups, of which we were previously unaware.

Chapter 10

Further Work

In this chapter we discuss further work that we believe would be useful. Our study has highlighted a number of areas where we suggest further work and we include them here.

10.1 Improving CSP programming skills

We believe our results could be improved dramatically with more expertise in formulating CSPs. We would seek to plug this gap and continue our work to improve this system.

10.2 Creating all_different Constraints

We have performed an initial study into deducing all_different, or other direct relationships between variables, from theorems by querying prolog with combinations axioms and conjectures regarding the equality of variables. We believe it is an achievable task to automate but more work is required to implement it.

10.3 Induced Constraints

In [2], the conjectures produced by HR were categorised into two types. Implied constraints, which were true for the entire domain, and induced constraints, which covered only a sub-set of the domain. Properly translated implied constraints, being true of the entire domain, can be added as constraints to the basic solver model without losing any solutions. Induced constraints, are only true for a sub-set of a domain and cannot therefore be added to the domain without potential loss of generality. However, if only a single example of a particular domain is required then [2] showed that induced constraints can be useful where they improve the effectiveness of the solver model without reducing the number of solutions to zero. The consideration of induced constraints has yet to be automated. In addition, concepts identified by HR may be used to sub-divide a problem search space, or provide streamlining which allows us to identify solutions more easily.

10.4 HR configuration

In [2] the configuration of HR was reviewed extensively to identify which settings would produce conjectures that were most suitable to being turned into constraints. Such a review, using the automated system, has yet to be performed.

10.5 Other Domains

We have studied only a small number of related domains. However, the approach we have developed is generic and should be applicable to many domains. We would like to continue investigating other areas to see if we can improve our constraint formulations.

10.6 Constraint Performance vs Order

In our analysis of the results, we have described the mechanics of how we believe different interesting theorems improve a solver's ability to solve the problem. We noticed that, in many cases, the effectiveness of a constraint will vary with the order of the solution. Take, for example, the theorem that states that the main diagonal of a QG3 quasigroup should be all different. We would expect that this constraint would become less effective as the order of the quasigroup increases and the main diagonal elements, as a proportion of the total elements, reduces. Ideally, we would like to be able to provide more information of this nature about constraints by considering how their effectiveness changes. This would be a useful addition to the system that could allow the user to restrict their search to constraints that will remain effective over more of the domain.

10.7 Completing The Cycle

[2] suggested that the four-phase cycle described in §2.6 could be completed by feeding the results of the reformulations back into HR for further processing, thereby completing the cycle. This has yet to be automated.

Bibliography

[1] S. Colton. *Automated Theory Formation in Pure Mathematics.* Springer-Verlag UK, 2001.

[2] Simon Colton and Ian Miguel. Constraint generation via automated theory formation. In *Proceedings of the Seventh International Conference on the Principles and Practice of Constraint Programming*, Cyprus, 2001.

[3] A. M. Frisch, C. Jefferson, B. Martynez Hernandez, and I. Miguel. The rules of constraint modelling. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, 2005.

[4] C Gomes and M Sellman. Streamlined constraint reasoning. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming*, 2004.

[5] Franois Laburthe. Choco: implementing a cp kernel. In *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, Singapore, September 2000.

[6] Carlsson M., Ottosson G., and Carlson B. An open-ended finite domain constraint solver. In *Proc. Programming Languages: Implementations, Logics, and Programs*, 1997.

[7] W. McCune. The Otter user's guide. Technical report, ANL, 1990.

[8] W. McCune. A Davis-Putnam program and its application to finite first-order model search. Technical report, ANL/MCS-TM-194, 1994.

[9] Casey Smith, Carla Gomes, and Cesar Fernandez. Streamlining local search for spatially balanced latin squares. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, 2005.

Appendix A

ICaRuS User Manual

A.1 Application Overview

We developed an application to allow a user to set up and test particular domains of interest, according to the process described in the main report.

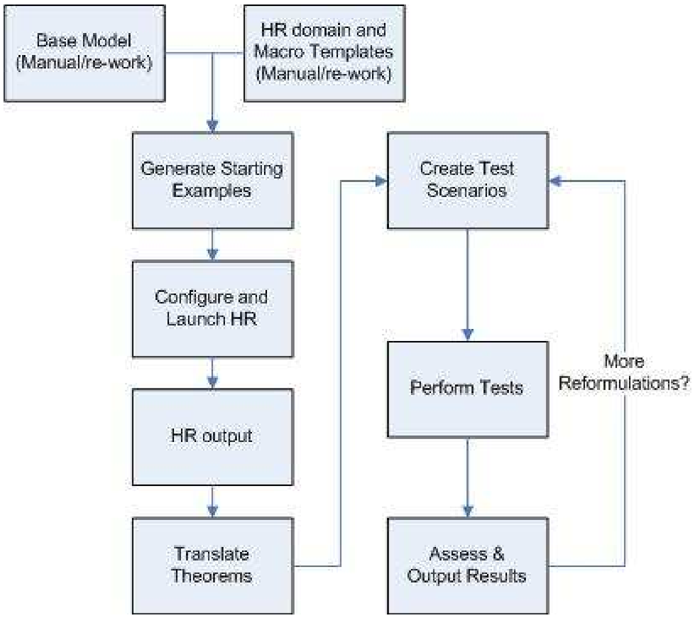


Figure A.1: Application: Process Overview

In many ways the application is simply an overlay for the underlying Prolog processor and efforts have been made to make the application as general, i.e. not domain specific, as possible.

A.1.1 Starting the Application

There are two modes of operation.

- user interface mode allowing for interactive testing and other investigation, or,
- batch mode, where the entire process is run automatically.

The easiest way to configure the application for batch running is to start the application in user interface mode, set all the required parameters and save the configuration. When the batch is invoked, these configuration settings will be used. The command for running the application in batch mode is

java -jar icarus *domainfilename*

See below for details of the domain file.

A.1.2 Stopping the Application

The application can be shut down by clicking the top right-hand corner close button. This will save the current configuration of the application, which will be reloaded again when the application is next started. In batch mode, the application will automatically close down after completing a batch or if there are errors.

A.1.3 Application properties

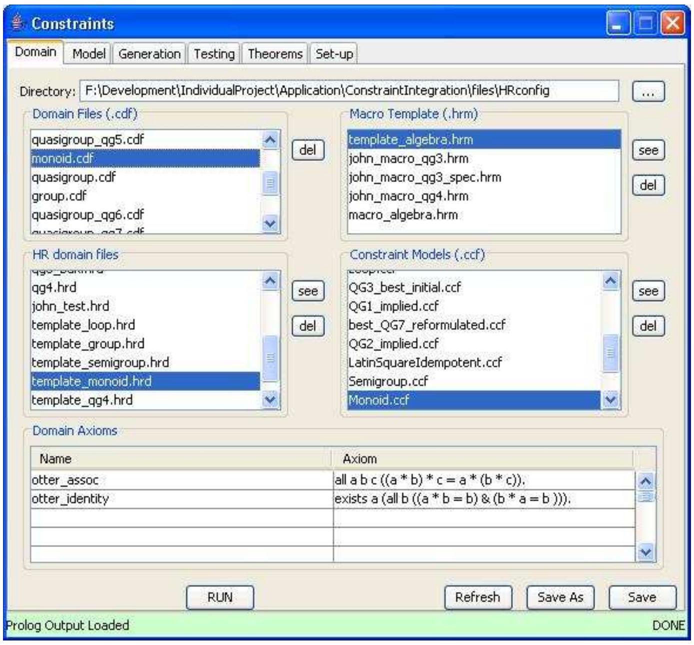
The settings of the application are saved, between invocations, in a config.ini file in the /config directory. They are saved as text strings and so can be easily modified without starting the application. However, it is easier to start the application in UI mode to modify parameters.

A.2 Main GUI

When the user starts the application, they are presented with a number of panels, each dedicated to a particular application function or step in the process.

A.2.1 Domain Panel

The domain panel is used for setting up the particular domain of investigation and for starting the HR process to search for conjectures. The HR configuration templates are selected here and the axioms to be used to prove conjectures are stated. A particular domain set-up consists of a HR macro file template, a HR domain file template, a base model and a list of axioms to be used by HR's theorem prover.



Directory: F:\Development\IndividualProject\Application\ConstraintIntegration\files\HRconfig

Domain Files (.cdf)

- quasigroup_qg5.cdf
- monoid.cdf
- quasigroup.cdf
- group.cdf
- quasigroup_qg6.cdf

Macro Template (.hrm)

- template_algebra.hrm
- john_macro_qg3.hrm
- john_macro_qg3_spec.hrm
- john_macro_qg4.hrm
- macro_algebra.hrm

HR domain files

- qg4.hrd
- john_test.hrd
- template_loop.hrd
- template_group.hrd
- template_semigroup.hrd
- template_monoid.hrd
- template_qg4.hrd

Constraint Models (.ccf)

- QG3_best_initial.ccf
- QG1_implied.ccf
- best_QG7_reformulated.ccf
- QG2_implied.ccf
- LatinSquareIdempotent.ccf
- Semigroup.ccf
- Monoid.ccf

Domain Axioms

Name	Axiom
otter_assoc	all a b c ((a * b) * c = a * (b * c)).
otter_identity	exists a (all b ((a * b = b) & (b * a = b))).

Prolog Output Loaded DONE

Figure A.2: Application: Domain Panel

- *Domain Configuration* Domain configuration files (.dcf) are used to store previously set-up domain configurations. They can be selected using the top left list box. New domain configurations can be saved allowing old configurations to be modified and re-used. The domain configuration files are stored in XML format and could be edited manually if so desired although it is much easier to start the application

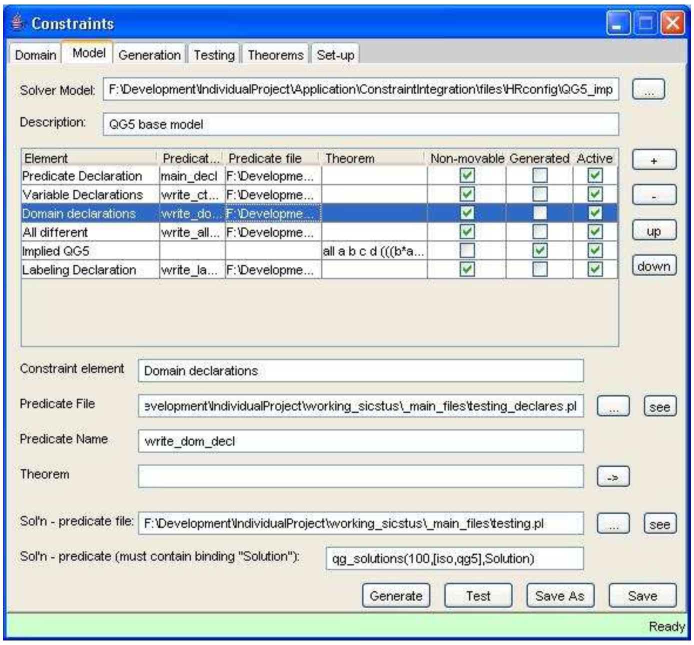
- *HR Configuration File* The template files that should be used in running HR are selected using the list boxes on this panel. The HR domain and macro files must be edited manually using a text editor - existing files can easily be modified to create templates for new domains. A template file for a macro file must contain tags to allow the application to correctly configure them. The macro file must contain a "[DOMAINFILE]" tag to allow for the correct HR domain file to be referenced, it should also include "// AXIOM-START //" "// AXIOM-END //" tags to allow for the prover axioms to be correctly entered and the "[AXIOMLIST]" tag for where the prover axioms are to be stated. The HR domain file template must contain the tags %EXAMPLES-START and %EXAMPLES-END to indicate where the positive examples of the domain will be placed when the HR configuration files are created. There are buttons to allow for these files to be inspected and buttons to delete any unwanted templates.

- *Constraint model* The base model used in the investigation and to generate initial examples is selected from the bottom right hand list box. A button allows the model to be viewed in the model panel, see below, and another to delete any unwanted model files.
- *Domain axioms* The axioms of the domains can be edited directly in the list box at the bottom of the panel.
- *Configuration directory* - All the HR configuration files are normally kept in the same directory, which can be selected using the text box and button at the top of the panel.

A.2.2 Model Panel

The Model pane allows the user to set up a model for solving a CSP. A CSP model is made up of a number of different statements, for instance variable and domain declarations. This panel allows the user to indicate which prolog predicates, from particular prolog source files, or which first order logic statements should be used for generating the various elements of a CSP model. Note that the predicates referred to aren't the actual CSP solver statements themselves but predicates that generate constraint statements by writing them to redirected standard output. This extra layer of redirection allows for configuration on the prolog side for such things as domain size changes, without having to amend the model set-up.

- *Location* Solver model files can be saved in XML format. The path to the currently displayed model is shown in the text box and the button to the right is used to load existing models.
- *Description* A text description of the model is entered here.
- *Main Table* Each of the rows in the main table relates to one particular solver element generator. Together, all the elements will generate the code to create a single Prolog source file which can be used to solve a CSP problem. For example, in the screen shot above, the first row refers to a predicate to write the main file declarations, i.e. a predicate that writes ":- use_module(library(clpfd)).", the first instruction in the generated Prolog source file, to use the built-in Sicstus CSP libraries. The predicate is located in the specified file. The "generated" checkbox allows users to indicate whether a particular element of the solver model should be translated from a first order logic theorem or from the stated predicates. Elements can be selected and deselected using the activated checkbox. The "movable" checkbox indicates which elements of the solver model can be moved when inserting new constraints and reformulating the model for testing. Buttons on the right hand side allow the user to add new elements, remove existing solver elements and also change their order.



Solver Model: F:\Development\IndividualProject\Application\ConstraintIntegration\files\HRconfig\QG5_imp

Description: QG5 base model

Element	Predicat...	Predicate file	Theorem	Non-movable	Generated	Active
Predicate Declaration	main_decl	F:\Developme...		☑	□	☑
Variable Declarations	write_ct...	F:\Developme...		☑	□	☑
Domain declarations	write_do...	F:\Developme...		☑	□	☑
All different	write_all...	F:\Developme...		☑	□	☑
Implied QG5			all a b c d (((b*a...	□	☑	☑
Labeling Declaration	write_la...	F:\Developme...		☑	□	☑

Constraint element: Domain declarations

Predicate File: evelopment\IndividualProject\working_sicstus_main_files\testing_declares.pl

Predicate Name: write_dom_decl

Theorem:

Sol'n - predicate file: F:\Development\IndividualProject\working_sicstus_main_files\testing.pl

Sol'n - predicate (must contain binding "Solution"): qg_solutions(100,[iso,qg5],Solution)

Figure A.3: Application: Model Panel

- *Editing elements* Any solver elements can be edited using the text boxes in the lower half of the panel. The paths to predicate files can be amended directly or selected with the button at the right hand side. A predicate file can be viewed to select a particular predicate. The "->" button transfers the first order logic statement to the Theorems Panel, as described in §A.2.5.

- *Solution Predicate* When the model is to be used to generate examples, then the output models should be in a format acceptable to HR and consistent with any definitions in the HR domain file. These text boxes indicate a wrapper predicate, which will take the solutions from the solver model and re-format them to be acceptable to HR. In the example above, the predicate is

  qg_solutions(100,[iso,qg5],Solution)

  which binds a list of solutions from the model to Solution. This particular predicate has parameters to indicate the maximum number of solutions should be limited to 100, that isomorphic examples should be ignored and that the example should be wrapped with qg5(. . .). The list bound to "Solution" is taken and added to the HR domain file when the configuration files are generated.

- *Buttons* The “generate” button generates a prolog solver formulation file, see §A.2.3. The “test” button runs a simple test for the model, see “Test Panel”, below. In addition, there are buttons to save the current status of the model pane as an XML constraint model configuration file (.ccf).

A.2.3 Generation

When the user presses the generate button, a Prolog source file is generated from the model shown on the Model Panel. The system steps through each of the rows of the table on the Model panel, which represent different elements of the solver model. If, for a particular element, the “generated” column is checked, then the system will translate the stated first order logic theorem, using the translation predicates from the Theorems Panel (§A.2.5). If the “generated” column is not checked then the system will use the predicate specified in the predicate and predicate file columns to generate Prolog source code. The Prolog source output file is stored in the file specified in the text box at the top of the panel, which will automatically be over-written if the user un-checks the checkbox at the lower right hand side. If the user has selected “display output”, then the results of the generation are echoed in the central text panel for review, deselecting this checkbox disables this function which is useful when the generated files are very large. If a Sicstus server is not running, or any other errors occur meaning the generation cannot be performed then, an error message is given.

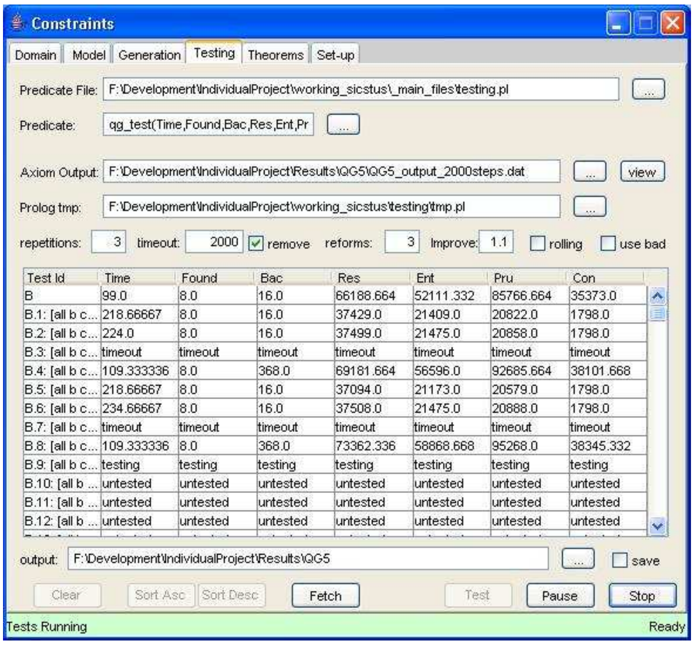
A.2.4 Testing Panel

The Testing panel displays the results of running tests on the base model either singly or by adding theorems from a specified theorem file. The testing process starts with the base model specified on the Model panel. If a single test has been selected, the test process will only test the base model. If a full test has been selected, the base model will be reformulated using the theorems in the specified output file according to the options selected.

- *Test Predicate* The actual test is performed by running the testing predicate indicated in the two text fields at the top of the panel. The columns displayed in the results table are automatically generated from the bindings in the test predicate. The testing process assumes that the first column of results, i.e. the first binding, is the parameter by which the results should be judged. This is normally the time taken to exhaust the search space but may easily be changed, to say backtracks, by changing the bindings of the testing predicate.

- *Temporary Output* A single test starts by generating the underlying model (either the base model or a reformulation). This file will be generated in the location specified by the “Prolog tmp” text box.

- *Repetitions* The results of a test can vary from one running to another therefore the user can specify how many times the same model should be



Predicate File: F:\Development\IndividualProject\working_sicstus_main_files\testing.pl

Predicate: qg_test(Time,Found,Bac,Res,Ent,Pr

Axiom Output: F:\Development\IndividualProject\Results\QG5\QG5_output_2000steps.dat

Prolog tmp: F:\Development\IndividualProject\working_sicstus\testing\tmp.pl

repetitions: 3 timeout: 2000 ☑ remove reforms: 3 Improve: 1.1 □ rolling □ use bad

Test Id	Time	Found	Bac	Res	Ent	Pru	Con
B	99.0	8.0	16.0	66188.664	52111.332	85766.664	35373.0
B.1: [all b c...	218.66667	8.0	16.0	37429.0	21409.0	20822.0	1798.0
B.2: [all b c...	224.0	8.0	16.0	37499.0	21475.0	20858.0	1798.0
B.3: [all b c...	timeout	timeout	timeout	timeout	timeout	timeout	timeout
B.4: [all b c...	109.333336	8.0	368.0	69181.664	56596.0	92685.664	38101.668
B.5: [all b c...	218.66667	8.0	16.0	37094.0	21173.0	20579.0	1798.0
B.6: [all b c...	234.66667	8.0	16.0	37508.0	21475.0	20888.0	1798.0
B.7: [all b c...	timeout	timeout	timeout	timeout	timeout	timeout	timeout
B.8: [all b c...	109.333336	8.0	368.0	73362.336	58868.668	95268.0	38345.332
B.9: [all b c...	testing	testing	testing	testing	testing	testing	testing
B.10: [all b ...	untested	untested	untested	untested	untested	untested	untested
B.11: [all b ...	untested	untested	untested	untested	untested	untested	untested
B.12: [all b ...	untested	untested	untested	untested	untested	untested	untested

output: F:\Development\IndividualProject\Results\QG5 □ save

Tests Running Ready

Figure A.4: Application: Testing Panel

run. The times, indeed any numerical results, from the test predicate will be averaged in the results panel, over the number of repetitions.

- *Timeout* The user can specify a timeout, in ms, after which the test will be stopped - this is useful for reducing overall testing times as the process doesn't have to wait for inefficient reformulations to finish before moving on. Note that the system automatically adjusts the timeout parameter whilst generating and consulting the model files as these processes may actually take longer than the tests themselves. Tests that have timed out are shown as "timeout" in the results and can be automatically removed from the results by selecting the remove timeout checkbox. Before running a full test, users should run some base model tests and decide an appropriate value for the timeout parameter.

- *Reforms* The user can limit the number of generations of reformulations in a test cycle by setting this parameter.

- *Improve* The user can specify the cut-off for accepting a model for further testing in the next generation. This number represents a proportion of the base model performance that a model must achieve before being considered. If the "rolling" text box is checked, then the best model performance to date is used in assessing whether models are put forward into the next round.

- *Use bad* If this check box is selected, then all theorems are used in making reformulations of the surviving models in a next round of testing. If this is unchecked, then only theorems that appear in the surviving models will be used.

- *Results* The results of the test are displayed in the table in the centre of the panel. The first column is the test id which indicates how the model has been created through the test process. The letter “B” is used to indicate the base model. Further reformulations have a *.n* suffix to indicate a reformulation previous model, and a list of the theorems that have been applied to the base model in preparing this formulation. For example, a test with the suffix B.3.14 : [*theorem1*,*theorem2*] is a 2nd generation test being the 14^{th} reformulation of the 3^{rd} reformulation of the base model and it uses *theorem1* and *theorem2*. The suffix numbers do not indicate the theorem being used as more than one reformulation may have arisen from the same theorem for different placements in the previous model. The remaining columns show the results from the bindings in the test predicate. After each generation of tests, the results are ordered using the first result column.

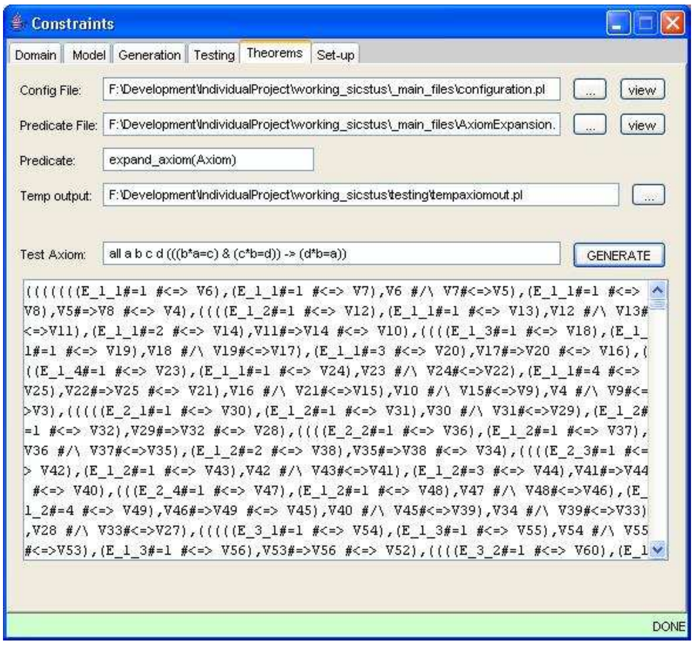
- *Output* The user can elect to save the results of any test run. The text box indicates where the results will be stored in a new directory identified with a timestamp for the test run. The directory will contain the base model configuration file, notes and results. The notes provide details of the test configuration, the different generations and timings. The results are stored in a results*n*.txt file, which holds the contents of the results table after each generation.

- *Buttons* There are buttons provided to start, stop and pause tests. The user may have to wait for the current test to complete when stop or pause is pressed as the application must wait for Sicstus to complete processing. The user must not stop and restart sicstus when pause is pressed as this may harm the current test configuration by losing the Sicstus server setup. The “clear” button clears the results screen. The “fetch” button will transfer the model from a selected test to the model pane, where it can be reviewed.

A.2.5 Theorems Panel

This is where the user specifies which predicates are going to be used to translate a given first order logic statements and allows the user to perform test translations of theorems.

- *Config File* This file is consulted by Sicstus before any other generation of prolog or running of tests. This allows the user to set configuration variables in the Prolog environment, such as the domain size to be investigated.



Config File: F:\Development\IndividualProject\working_sicstus_main_files\configuration.pl

Predicate File: F:\Development\IndividualProject\working_sicstus_main_files\AxiomExpansion.

Predicate: expand_axiom(Axiom)

Temp output: F:\Development\IndividualProject\working_sicstus\testing\tempaxiomout.pl

Test Axiom: all a b c d (((b*a=c) & (c*b=d)) -> (d*b=a))

```
(((((((E_1_1#=1 #<=> V6),(E_1_1#=1 #<=> V7),V6 #/\ V7#<=>V5),(E_1_1#=1 #<=>
V8),V5#=>V8 #<=> V4),((((E_1_2#=1 #<=> V12),(E_1_1#=1 #<=> V13),V12 #/\ V13#
<=>V11),(E_1_1#=2 #<=> V14),V11#=>V14 #<=> V10),((((E_1_3#=1 #<=> V18),(E_1_
1#=1 #<=> V19),V18 #/\ V19#<=>V17),(E_1_1#=3 #<=> V20),V17#=>V20 #<=> V16),(
((E_1_4#=1 #<=> V23),(E_1_1#=1 #<=> V24),V23 #/\ V24#<=>V22),(E_1_1#=4 #<=>
V25),V22#=>V25 #<=> V21),V16 #/\ V21#<=>V15),V10 #/\ V15#<=>V9),V4 #/\ V9#<=
>V3),(((((E_2_1#=1 #<=> V30),(E_1_2#=1 #<=> V31),V30 #/\ V31#<=>V29),(E_1_2#
=1 #<=> V32),V29#=>V32 #<=> V28),((((E_2_2#=1 #<=> V36),(E_1_2#=1 #<=> V37),
V36 #/\ V37#<=>V35),(E_1_2#=2 #<=> V38),V35#=>V38 #<=> V34),((((E_2_3#=1 #<=
> V42),(E_1_2#=1 #<=> V43),V42 #/\ V43#<=>V41),(E_1_2#=3 #<=> V44),V41#=>V44
 #<=> V40),(((E_2_4#=1 #<=> V47),(E_1_2#=1 #<=> V48),V47 #/\ V48#<=>V46),(E_
1_2#=4 #<=> V49),V46#=>V49 #<=> V45),V40 #/\ V45#<=>V39),V34 #/\ V39#<=>V33)
,V28 #/\ V33#<=>V27),(((((E_3_1#=1 #<=> V54),(E_1_3#=1 #<=> V55),V54 #/\ V55
#<=>V53),(E_1_3#=1 #<=> V56),V53#=>V56 #<=> V52),((((E_3_2#=1 #<=> V60),(E_1
```

DONE

Figure A.5: Application: Theorems Panel

- *Translation Predicate* Two textboxes indicate the location and name of the predicate that is to be used to translate first order sentences into constraints. The predicate should contain the binding parameter "Axiom" for the statement to be translated and should write the result to standard output, from where it will be redirected.

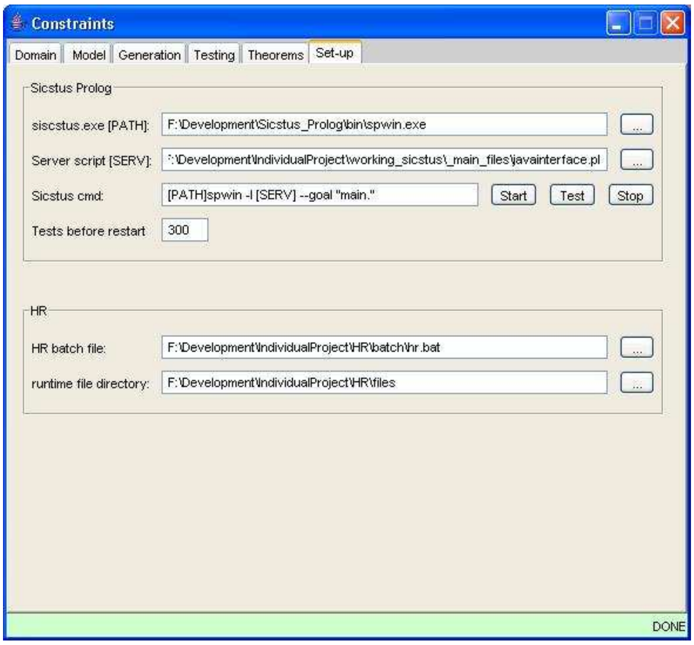
- *Temp output* This is the file where the output of the generated theorems will be stored during the translation process.

- *Test Axiom* This allows the user to specify a statement to be translated into prolog, the result is shown in the text panel below.

A.2.6 Set-up Panel

This panel is provided to allow the user to easily configure the application's interaction with external programs.

- *Sicstus* In order to have the application automatically launch Sicstus, the user must specify the path to the executable, a Prologbeans server script file and the command to be executed to launch Sicstus. Buttons are provided to start, test and stop a Sicstus server. In addition, the user can specify a number of tests after which Sicstus will be stopped and



Sicstus Prolog

siscstus.exe [PATH]: F:\Development\Sicstus_Prolog\bin\spwin.exe

Server script [SERV]: :\Development\IndividualProject\working_sicstus_main_files\javainterface.pl

Sicstus cmd: [PATH]spwin -l [SERV] --goal "main."

Tests before restart 300

HR

HR batch file: F:\Development\IndividualProject\HR\batch\hr.bat

runtime file directory: F:\Development\IndividualProject\HR\files

Figure A.6: Application: Setup Panel

re-started. This is useful to conserve resources when running the Sicstus server for long periods.

- *HR* If the user wishes to use automatic configuration and launching of HR they must specify the path to the HR batch file and the directory where HR macro and domain files will be placed.

A.3 Batch Mode

A user wishing to run the application in batch mode uses the following command from the command line:-

java - jar icarus *domainfilename*

The application will load *domainfile*. It will create examples from the solver model it refers to and generate HR's configuration files according to the domain file's parameters. The batch then launches HR in macro mode (i.e. no visible application) and waits for it to complete the number of steps stated in the macro file, whereupon HR will terminate and write its results to a specified file. At this point the system will start a full test using the base model and HR's output. The test results are automatically saved after each generation. The application exits when testing has been completed.