A Constraint Solver Synthesiser: Case for Support

Ian Miguel, Dharini Balasubramaniam, Ian P. Gent, Christopher Jefferson, Tom Kelsey, Steve Linton

November 5, 2009

Abstract

This document is the case for support for Grant number EP/H004092/1, funded by the EPSRC in the UK. Our **aim** is to improve dramatically the scalability of constraint technology, while removing its reliance on manual tuning by an expert. This aim will be realised through the development of a *constraint solver synthesiser*, the principal components of which are a *Model Analyser*, a *Constraint Solver Generator*, and a *Generator Tuner*. As a research proposal, the reader should bear in mind that this is a proposal for future work, rather than a report on completed research.

Acknowledgements

We very gratefully thank the Engineering and Physical Sciences Research Council (EPSRC) of the United Kingdom for their very generous support of the research proposed in this document, under grant number EP/H004092/1.

Case for Support

A Background

Constraints are a natural, powerful means of representing and reasoning about combinatorial problems that impact all of our lives. For example, in the production of a university timetable many constraints occur, such as: the maths lecture theatre has a capacity of 100 students; art history lectures require a venue with a slide projector; no student can attend two lectures at once. Constraint solving offers a means by which solutions to such problems can be found automatically. Its simplicity and generality are fundamental to its successful application in a wide variety of disciplines, such as: scheduling; industrial design; aviation; banking; combinatorial mathematics; and the petrochemical and steel industries, to name but a few examples [17].

Currently, applying constraint technology to a large, complex problem requires significant manual tuning by an expert. Such experts are rare. The central aim of this project is to improve dramatically the scalability of constraint technology, while simultaneously removing its reliance on manual tuning by an expert. We propose a novel, elegant means to achieve this: a *constraint solver synthesiser*, which generates a constraint solver specialised to a given problem. Synthesising a constraint solver tailored to the needs of an individual problem is a groundbreaking direction for constraints research, which has focused on the incremental improvement of general-purpose solvers. Synthesising a solver from scratch has two key benefits, both of which will have a major impact. First, it will enable a fine-grained optimisation not possible for a general solver, allowing the solution of much larger, more difficult problems. Second, it will open up many exciting research possibilities. There are many techniques in the literature that, although effective in a limited number of cases, are not suitable for general use. Hence, they are omitted from current general solvers and remain relatively undeveloped. The synthesiser will, however, select such techniques as they are appropriate for an input problem, creating novel combinations to produce powerful new solvers. The result will be a dramatic increase in the number of practical problems solvable without the input of a constraints expert.

Constraint Modelling and Solving: The State of the Art. Constraint solving of a combinatorial problem proceeds in two phases. First, the problem is *modelled* as a set of *decision variables*, and a set of *constraints* on those variables that a solution must satisfy. A decision variable represents a choice that must be made in order to solve the problem. The *domain* of potential values associated with each decision variable corresponds to the options for that choice. In our timetabling example one might have two decision variables per lecture, one representing its time and the other its venue. The second phase consists of using a constraint solver to find solutions to the model: assignments of values to decision variables satisfying all constraints (e.g. a valid timetable). Constraint solvers typically employ a systematic backtracking search through the space of partial assignments in order to find solutions.

Constraints research faces a major challenge: to deliver constraint solving that scales easily to problems of practical size. Current constraint solvers, such as Choco, Eclipse, Gecode, Ilog Solver, or our own Minion are *monolithic* in design, accepting a broad range of models. This is convenient, but at the price of a necessarily complex internal architecture, resulting in significant overheads and inhibitting efficiency and scalability. The complexity of current solvers also means that it is often prohibitively difficult to incorporate new techniques as they appear in the literature. This is a significant disadvantage as it is this solver "inertia" that largely dictates the direction of the field. A further drawback is that current solvers perform little or no analysis of an input model, so individual model features cannot be exploited to produce a more efficient solving process. To mitigate these drawbacks, constraint solvers often allow manual tuning of the solving process. This requires considerable expertise, preventing the widespread adoption of constraints as a technique for solving the most challenging combinatorial problems.

Propositional satisfiability (SAT) offers a stark example of the effectiveness of tailoring solvers to particular problems. A SAT problem is a constraint problem in which all variables are boolean, and constraints are given in conjunctive normal form. Modern solvers, such as MiniSat exploit this structure to make search extremely efficient. Hence, SAT solvers are orders of magnitude faster than constraint solvers applied to the same type of model, and are used to solve large, complex SAT problems of practical interest, such as chip verification. Although monolithic in design, our Minion solver provides strong evidence that a careful approach to design and implementation can also have a major impact on constraint solving. Minion has been shown to be orders of magnitude faster than competing state-of-the-art constraint solvers [2].

Redefining the State of the Art. Constraint solving has great potential for widespread use in solving problems of real importance, but is inhibited by the compromises made in monolithic solvers. By generating a solver dedicated to a particular model, this project will unlock that potential by greatly increasing efficiency.

Solver synthesis is summarised in Figure 1. It begins with a novel *model analysis* step to determine which features of a constraint solver are, and are *not*, necessary to solve an input model efficiently. For example, a monolithic solver might support various domain types, such as Booleans, integers, or domains represented by their bounds only. Supporting multiple domain types increases the internal complexity, particularly the constraint *propagation* algorithms that make inferences based on the constraints in the model and the current state. Suppose that a given model requires bounds variables only; stripping out the superfluous variable types will enable considerable streamlining of the solver. Constraint propagation usually records inferences as reductions to the domains of decision variables. Suppose also that, for some variables, it can be determined that only the upper bound will ever be updated by constraint propagation. The representation of these variables can be simplified still further by stripping out functionality required to revise the lower bound. This is a finer-grained optimisation than any current solver considers.

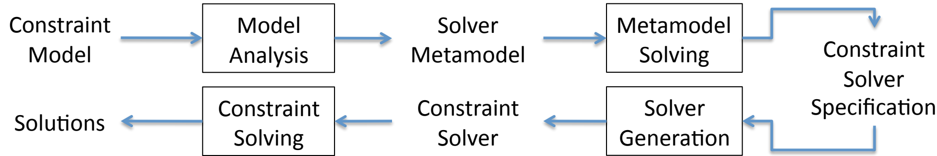

Figure 1: Constraint Solver Synthesis: increasing efficiency and removing manual tuning.

A further limitation of current solvers is in their fixed (and often) opaque choice of representation for individual variable types. Generally, for example, two fundamentally different representations are used for the domain of an integer variable: a bit array for possible values indicating their presence or absence; or a list of ranges of values such as `{[1..5],[8..10]}`. Current solvers offer little or no choice in this, and often hide their internal representation. There are many more aspects of representation, which can have significant effect on performance, each with their own tradeoffs. No monolithic solver can support all combinations. A synthesised solver, however, can be optimised for a particular problem, with different choices made for different variables.

Domain representation is just one component of a solver. Other significant components include: the constraint propagators and the queue that organises them, the search procedure, and management of restoration of state on backtracking. Each major component has sub-components that can be optimised in special cases. Considering solver construction at this level of detail enables the incorporation, as appropriate, of some undoubtedly powerful techniques that are not usually implemented because they have significant costs and are not always useful. These include: conflict recording [11]; backjumping [13]; many constraint propagation schemes such as singleton arc consistency [6] and neighbourhood inverse consistency [8]; and advanced work on exploiting symmetry [10]. By generating specialised solvers, we can use these techniques when beneficial, with no overhead when they are unnecessary.

Constraint solver synthesis will also expose research issues often hidden behind the facade of current solvers and their need to make a single choice on key design decisions. One example is the constraint queue, responsible for scheduling constraint propagation. It has long been known [**?**] that the order in which constraints are propagated can have a significant effect on overall efficiency. However, research in this area is hampered because, in current constraint solvers, the queue is either inaccessible, or prohibitively complex to modify. The solver generator will select a queuing scheme appropriate to the constraint solver generated, for example balancing the cost against the strength of propagation of the constraints in the queue. Similar concerns affect every aspect of the construction of a constraint solver, and will lead to many new insights into important research questions in constraint solving. Equally importantly, the synthesiser will be designed to be extensible, fostering new research in this and other areas of constraint solving neglected because of solver inertia.

Once model analysis is complete, the next step is to formulate a constraint solver specification, which will comprise a set of compatible components describing a constraint solver tailored to the input model. Not all combinations of components are compatible, for example a constraint propagator optimised for one variable type may be inefficient or even incorrect if used with another. Hence, solver specification is itself a natural constraint problem. We will refer to the problem generated by model analysis as the *metamodel*, which could be solved by any constraint solver (although we propose to use an elegant bootstrapping process in which a specialised metamodel solver is produced via the synthesiser). The variables of the metamodel represent the solver components for which choices are required (e.g. variable types supported by the synthesised solver) and the domains are the options for those choices. Model analysis will naturally reduce the options for these choices, such as ruling out support for variables that allow the revision of both upper and lower bounds if only the revision of the lower bound can take place. Constraints will ensure only compatible combinations of components are selected.

The *solver generator* is responsible for taking the solution to the solver metamodel (corresponding to a solver specification) and producing a highly-optimised constraint solver. This is a complex task, involving choosing and integrating the best algorithms and data structures to implement the specified combination of components. To meet this challenge, we propose a *generative* approach, described in the methodology section below.

Ambitious but Achievable. Our research is highly ambitious. We seek to reinvent the way that constraint solvers are built. Our programme of constructing a *constraint solver synthesiser* will enable the rapid production of specialised solvers incorporating the latest powerful constraint techniques. This will make constraint solving much more important in practice, with applications in many more domains than at present. Our ambitious research is also achievable, due to our experience in building a state-of-the-art constraint solver and our proposed methodology and research programme. Experience with Minion has led us to identify the key research issues facing us, to which we will apply the well-understood methodology of generative programming.

B Programme and Methodology

B.1 Aim and Objectives

Our **aim** is to improve dramatically the scalability of constraint technology, while removing its reliance on manual tuning by an expert. This aim will be realised through the development of a *constraint solver synthesiser*, the principal components of which are a *Model Analyser*, a *Constraint Solver Generator*, and a *Generator Tuner*.

The project **objectives** in detail are to:

1. Develop a *Model Analyser* that, given a constraint model, produces a constraint solver *metamodel* whose solution corresponds to a specification for a constraint solver tailored to solving the input model.
2. Develop a *Constraint Solver Generator* that, given a constraint solver specification comprising interconnected solver components drawn from a component library, produces an optimised constraint solver.
3. Develop a *Generator Tuner* that, via sample instances of a problem class, identifies the critical components of a candidate solver for that class and revises the metamodel to prioritise those components, improving the solver further.
4. Create *demonstration applications*, using the synthesiser to address challenging problems previously out of reach of constraint solving.
5. Evaluate thoroughly the constraint solver synthesiser on a diverse set of benchmarks.

B.2 Methodology

The complexity of the challenge we address demands the use of a solid, well-understood methodology. Hence the choice of *generative programming* [7], a software engineering paradigm based on modelling software *families* (in this case, a family of constraint solvers). Given a requirements specification, generative programming enables the production of a highly customised and optimised system, generated automatically from a component library. Our experience with the Minion constraint solver will be invaluable in building this library. The advantages of generative programming over, say, dynamic linking of libraries are threefold. First, performance, and therefore scalability, will benefit from the leaner interfaces and tighter coupling of components that the generative approach affords. Second, comprehension: configuration knowledge governing how solver components can be combined is captured in a language designed for the purpose, rather than the language designed for implementing the system. This supports the third advantage, extensibility: generative programming is designed to support the integration of new components, which will be crucial in fostering new constraints research within the synthesiser framework.

To construct a generative framework, we will build on Minion. Its speed is due in large part to the use of C++ templates. We have a small number of distinct variable types, such as Booleans and discrete integers. Using templates, the compiler automatically builds each constraint for every combination of variable types. This avoids virtual function calls and allows each propagator to be optimised for its specific inputs. Unfortunately, compile time grows as a power of the number of variable types, limiting us to a single digit number of variable types. This simultaneously shows the value of templating but also its limits for a monolithic solver. Through a generative approach, we can expand the range of variable types, essentially without limit. We can include specialised types not normally cost-effective to include in Minion. Moreover, templates are a proven methodology for implementing generative programming [7]. The advantages of solver generation extend to fine grained details in every aspect of the solver, tuned to a particular problem's needs. Since it is already template-based, we will use Minion to build a prototype solver generator. This means we will be generating constraint solvers of world-class from a very early stage, and we always have a working synthesiser.

As the workplan shows, we have adopted an iterative development strategy and a rigorous evaluation schedule with explicit milestones. Design and implementation phases will be supported by extensive testing and empirical analyses throughout the project. These will confirm that the approaches taken are successful, or help suggest remedies for any failings uncovered.

To identify the critical components of a generated solver, we will gather profiling data on its operation during search, instrumenting each component in the library appropriately. This information will support both performance tuning and evaluation. Through the nature of generative programming, instrumentation will incur no cost when unused.

B.3 Research Programme

Task 1: The Model Analyser

Task 1 is to develop a *Model Analyser* to determine the best combination of solver components for an input model.

WP1.1: Model Analysis The input constraint models will be expressed in the constraint modelling language ESSENCE$'$ [9]. Given an ESSENCE$'$ model, a range of analyses will be performed to determine the constraint solver features necessary to solve the model most effectively. It will be relatively straightforward to determine that a certain type of variable is not present in the model, and so does not need to be supported by the solver. A more detailed analysis will be required to determine that, say, only the upper bound of a variable will be modified by constraint propagation. There are many subtle decisions and details that have a profound effect on constraint solving, such as the style of propagation queue (Task 2.2), or the method of restoration of state upon backtracking (Task 2.4). Such decisions may be difficult initially, but they will be informed by experience gained from empirical testing.

WP1.2: Generic Solver Metamodel The solver metamodel describes the problem of finding a compatible combination of components to specify a constraint solver. Its variables represent features or sub-features of a constraint solver, such as queuing style (see WP2.2), and their domains represent the various solver components developed in Task 2 capable of providing that feature, such as a simple queue or a two-level priority queue. Metamodel constraints reflect compatibilities among the elements of the component library, hence its solutions correspond to valid solver specifications. WP1.2 is to construct the *generic* metamodel, which describes the compatibilities among the *entire* component library, and hence encompasses all possible options for solver synthesis. The generic metamodel will be adapted to the task of synthesising a solver suggested by model analysis in WP1.3.

WP1.3: Specialising and Solving the Metamodel Dependent on the results of model analysis (WP1.1), we will automate the *specialisation* of the generic metamodel (WP1.2) to describe the problem of specifying a solver for

a particular input model. Compromise will usually be necessary among the components suggested by model analysis, so a key part of specialisation will be to add an *objective function* to the generic metamodel, to allow optimisation. Further examples of specialisation include removing metamodel variables for configuring features that model analysis suggests should not be part of the synthesised solver, and also removing metamodel domain elements corresponding to components that model analysis has ruled out (e.g. certain variable types). The specialised metamodel is itself a constrained optimisation problem, so it can be solved by a constraint solver. Its complexity will depend on the component library of the solver generator, but it is not expected to be difficult to solve. Initially, we will use Minion to solve the metamodel. As the synthesiser develops, it will be used to *bootstrap* this process, synthesising a solver specifically for the metamodel.

Task 2: The Solver Generator

Task 2 is to develop the *Solver Generator* that takes a constraint solver specification, embodied by the solution to the metamodel, and produce an optimised constraint solver.

WP2.1: Solver Generation Component selection is performed by the solution of the metamodel from Task 1, guaranteeing that a compatible set of components have been selected. It remains to integrate these components efficiently to produce an optimised constraint solver. This is a complex task, and is the responsibility of the constraint solver generator, the focus of this workpackage. Using the aforementioned generative architecture, the solver generator will make informed choices and tradeoffs about the best low-level data structures and algorithms to realise the components specified. For example, if restoration of state is to be performed by simple block copying, the generator may decide to pay the one-off cost of locating all backtrackable state in a single block of memory to facilitate the copying process, which will be performed many times.

WP2.2: Component Library: Variable Types The fine control of the generative approach affords flexibility in the services that a variable provides to the constraint propagators and search process. For instance, a popular search heuristic is known as *smallest-domain* [14], in which the search process selects the variable with the smallest remaining domain to assign. To support this heuristic, the solver must maintain this information throughout search, which incurs an overhead. When this information is not used, we can avoid this overhead by not recording domain size and thus improve the efficiency of the solver.

WP2.3: Component Library: Constraint Propagation and Propagation Queue There is extensive research on different styles and strengths of constraint propagation for different situations. We will use this research to populate the propagator component library. One decision is whether the constraint store should be dynamic (allow constraints to be added during search) or static. The constraint propagation style that is most often used records inferences simply as reductions to the variable domains. There are, however, many propagation styles that can record inferences as new constraints, such as *path* consistency [12]. The queuing mechanism, which is responsible for organising the order in which constraint propagators are invoked, is known to have an important effect on efficiency [15]. The queue components will encapsulate a variety of queueing mechanisms, from simply assuming that each propagator has equal priority to complex multi-level queues in which more expensive propagators are given a low priority.

WP2.4: Component Library: Search Strategies It is well known that good search strategies and heuristics play a vital role in efficient constraint solving. The solver generator allows us to support a wide variety of such search strategies, including those used infrequently. A principal component of any constraint solving search strategy are the heuristics that decide which variable, and which value in the domain of that variable, to assign next. The component library will contain a variety of heuristics, from the least to the most informed in the literature. Some require a greater overhead than others in terms of the information required to compute the heuristic, such as the smallest-domain example given above, and hence interact with the variable components. The library will also contain various powerful, but neglected techniques, such as backjumping [13] (which supports backtracking over multiple decisions) and conflict recording [11] (which records the reasons for dead ends in the form of new constraints).

WP2.5: Component Library: Restoration of State When a dead end is reached during search, a constraint solver *backtracks* and tries another option. A key part of the backtracking process is restoration of state. One option is to make a copy of the state before each branching decision, and then restore the copy upon backtracking. If the state is large, this can be expensive, but for problem classes with small state, this is a viable option. Another approach, which is more appropriate when the state is large, is to record only the parts of the state that change, and restore them upon backtracking. As per our Minion solver, we will also employ watched literals [3], a technique adopted from propositional satisfiability that can drastically reduce the amount of state restoration required.

Task 3: The Synthesiser Tuner

Most constraint models describe a parameterised problem *class* (e.g. the class of sudoku puzzles). For input to a constraint solver, an instance of the class is obtained by giving values for the parameters (e.g. the pre-filled cells on the sudoku grid). Hence, it is natural to synthesise a solver for a problem class. Having synthesised a solver for a problem class, the *Synthesiser Tuner* will profile its performance on sample instances to identify critical components, use the results to revise the metamodel, and so synthesise an improved solver. The additional work in tuning can be amortised over the problem class.

WP3.1: Profile Analysis. Given an instrumented solver, we will select test instances of the problem class for which the solver is generated. We will be careful to select diverse instances so as to test the solver thoroughly and form a balanced view of performance. We will automate the process of analysing the performance data to identify the critical components of the solver on these benchmark instances in preparation for metamodel revision (WP3.2).

WP3.2 Metamodel Revision. By prioritising efficient implementation of critical solver components identified in the analysis of WP3.1 over non-critical components, solver performance can be further improved. The natural way to produce a solver specification that reflects these priorities is to modify the original solver metamodel. We will update the constraints and objective function so that an optimal solution obtained by re-solving the metamodel will correspond to a specification with the desired priorities. In this workpackage we will automate this process.

WP3.3: Iterated Tuning. A tuned solver can itself be profiled and analysed. We will use the synthesiser tuner iteratively to progressively hone a solver using two or more phases of synthesis and profiling. Furthermore, if tuning has had the opposite effect to that desired (i.e. performance degradation), perhaps as a result of insufficient test data, iterated tuning offers the opportunity to compare the analyses of two (or more) solvers to gain a yet clearer picture of the key decisions in synthesising a solver for a given problem class.

WP3.4: PhD Support. Task 3 will be the subject of a PhD studentship supervised by Ian Miguel. We include here PhD support which does not otherwise contribute to the project. As well as normal supervisions and compliance with departmental procedures at St Andrews, we have laid aside two substantial periods of time. At the start of the project the student will undertake a literature review in relevant aspects of constraint programming and software engineering. Time has been put aside for writing up.

Task 4: Demonstrator Applications

Task 4 is to create applications to test the synthesiser during development, and to demonstrate how it can solve problems previously out of reach of constraint solving.

WP4.1: Applications We will require applications with which to test the synthesiser and its components throughout the project. We already have a large benchmark suite for Minion, containing various important problem types such as planning, scheduling and packing, but will enlarge this throughout the project for evaluation purposes.

WP4.2: Demonstrators For communication and outreach to other fields, we will identify a much smaller suite of *demonstrator* problems drawn from diverse areas, such as bioinformatics, combinatorial mathematics, and industrial design, planning and scheduling. We will then test whether the synthesiser can produce constraint solvers powerful enough to solve them.

Task 5: Evaluation

We will evaluate the synthesiser throughout development. This will provide feedback on early prototypes and a significant body of final empirical results.

WP5.1: Evaluating the Model Analyser. This workpackage will evaluate our success in creating an automated model analyser capable of determining the combination of solver components best suited to solving an input model. For each of a benchmark set of ESSENCE′ models, we will compare the solver specification produced, as expressed in the solution to the solver metamodel, against a hand analysis produced using our own expertise. Where the two differ, we will examine whether the hand analysis is itself an alternative solution to the metamodel (in which case it might be desirable to alter the objective function of the metamodel), or whether the hand analysis had a deeper insight into the model (in which case we must consider improvements to the model analyser).

WP5.2: Evaluating the Solver Generator. This workpackage will evaluate our central hypothesis: that highly-specialised constraint solvers can significantly outperform existing state-of-the-art monolithic solvers. We will use a benchmark set of ESSENCE′ models, synthesising a solver for each. The synthesised solvers will be compared against existing solvers applied to the same model. Here we will exploit the fact that ESSENCE′ is solver-independent, and hence an ESSENCE′ model is suitable for input to a variety of constraint solvers with few significant modifications.

WP5.3: Evaluating the Synthesiser Tuner. This workpackage will evaluate the extent to which the synthesiser tuner can improve a synthesised solver. For each of a benchmark set of models, we will compare the solver synthesised from that model without tuning against the tuned version of that solver. We will experiment with the size and composition of the set of training instances used for tuning, as well as the utility of iterated tuning, in which a tuned solver is itself profiled and analysed to see if further improvements can still be made.

B.4 Timeliness and Novelty

To the best of our knowledge, the research herein is completely novel. The closest point of comparison is the G12 project [16]. Although this project also seeks to generate a solver for an individual problem, the approach taken is to combine *existing* general solvers from constraints and related fields like propositional satisfiability and operations research into a hybrid. This is a quite different challenge: to enable solvers from different disciplines to communicate effectively. The timeliness of this project lies in the maturity of the Minion constraint solver, from which we have learned a great deal about the importance of careful design and implementation of a constraint solver.

B.5 Management, People and Development, Risk

The Principal Investigator, Ian Miguel, is an experienced researcher who will lead the project. Ian Gent and Steve Linton have managed a number of grants and will assist with the management process. Dharini Balasubramaniam will provide her expertise on software engineering in general and generative programming in particular. Tom Kelsey will bring to bear his experience of applying Minion to challenge problems. Overall, the team has extensive experience of managing successful grants. Regular physical meetings will ensure synchronisation of the various project threads. We have also set a milestone at every six months to help us to monitor our progress. Christopher Jefferson will take on the role of software architect and act as team lead for the software aspects of the project.

This project will be an outstanding opportunity for the development of the staff and student employed. Apart from working in a world-class research group, benefits will include extensive contact with the investigators with their different specialist skills. An important aspect of develop-

ment will be working on a project bringing together constraint programmers and software engineers. The student will particularly benefit from working on a focused project with postdoctoral researchers as well as academic staff.

Our research is very ambitious and carries risk. Our methodology of an iterative cycle of milestones is a part of our risk management strategy. This will help us identify problems early. If we discover fundamental problems with our programme, we will be able to redirect our research efforts to the interesting questions that would surely arise. In our workplan we categorise deliverable software at milestones as one of: a prototype, which may not be functionally complete; an alpha, which will be almost complete but may be inefficient; a beta, which should have close to final algorithms; and a final version.

C Relevance to Beneficiaries

Industrial Community Constraint programming has found success in the industrial community in diverse applications such as aviation, banking and the petrochemical and steel industries. However, this success is inhibited by the lack of scalability of off-the-shelf constraint solvers. Applying constraint technology to a complex industrial problem currently requires the knowledge of a constraint expert. The prohibitive cost of employing such experts, and the fact that they are few in number, prevents constraint programming from becoming truly widespread. The constraint solver synthesiser proposed by this project will enable users with far less experience to model and solve complex problems. This will lead to a far greater proportion of the industrial community being able to exploit efficient, scalable constraint technology.

The research team has extensive formal and informal industrial links with important industrialists in the constraints field, such as Jean-Francois Puget (ILOG/IBM), Youssef Hamadi (Microsoft Research) and Andrew Davenport (IBM). We have collaborated with all three. While we will maintain industrial links, we have not formed a formal collaboration for this project. No one industrial application is our target. While it might seem appropriate to work with a constraint technology company such as ILOG, we have in the past been told that they regard us as competition, which is a testament to the quality of our solvers but reduces the chance for collaboration.

Academic Community There are two main groups of academic beneficiaries: the constraints community, and the wider community with combinatorial problems to solve. The constraints community will benefit from the availability of new techniques for synthesising constraint solvers, and especially from the ability to integrate new techniques. This will greatly benefit research, as new ideas can be prototyped rapidly in a production quality constraint solver, even if they have features incompatible with current solvers. A wider use of constraints throughout academia will also benefit the constraints field through new applications, ideas, and collaborations. Outside the constraints community, researchers will benefit from the ability to construct powerful solvers for combinatorial problems without having constraints expertise, and without the overheads imposed by monolithic solvers. We have found our work of particular benefit to combinatorial mathematicians, who have been able to find new structures of interest to them through the use of our constraint solvers, as shown for example by Kelsey's semigroups work using Minion.

D Dissemination and Exploitation

We will ensure that our work is disseminated as widely as possible. We will publish workshop, conference and journal papers, taking advantage of the feedback received at each stage to polish our work. Where appropriate, we will participate in workshop series, such as those concerning constraint modelling and solving, to present our early results to the most relevant researchers. We will also target the highest quality international conferences in both AI (e.g. IJCAI, AAAI, ECAI, CP) and Software Engineering (e.g. GPCE, ASE), at which we already have a very strong record of publication, to maximise the impact of our work. We will submit mature work to the best international journals, such as Artificial Intelligence Journal and the Journal of Automated Software Engineering.

We will continue to exploit the web to disseminate our work to both the industrial and academic communities. This is a particularly appropriate avenue for the new software and benchmark sets that this project will produce, and one with which we have been very successful in disseminating our Minion constraint solver.

We will liaise with industrial contacts (above) to identify routes for transferring results into their business. The University Research and Enterprise Service's remit specifically includes technology transfer and licencing of research results. Should commercially exploitable results arise from the project, Research and Enterprise Services will be approached to help with the technology transfer.

Bibliography

1. A. Distler, T. Kelsey. The Monoids of Order Eight and Nine. AISC 2008: 61-76
2. I. Gent, C. Jefferson, I. Miguel. Minion: A Fast, Scalable Constraint Solver. ECAI, 98-102, 2006.
3. I. Gent, C. Jefferson, I. Miguel. Watched Literals for Constraint Propagation in Minion. CP, 182-197, 2006.
4. I. Gent, C. Jefferson, I. Miguel, P. Nightingale. Data Structures for Generalised Arc Consistency for Extensional Constraints *AAAI 2007*: 191-197. 2007.
5. I. Gent, I. Miguel, P. Nightingale. Generalised arc consistency for the AllDifferent constraint: An empirical survey. *Artificial Intelligence*: 172 (18):1973-2000, 2008.
6. C. Bessiere, R. Debruyne. Theoretical Analysis of Singleton Arc Consistency and Its Extensions. *Artificial Intelligence*, 172(1), 29-41, 2008.
7. K. Czarnecki, U.W. Eisenecker. Generative Programming: Methods, tools, applications, Boston, 2000.
8. E. Freuder, C. Elfe. Neighborhood inverse consistency preprocessing. AAAI, 202-208, 1996.
9. A. M. Frisch, W. Harvey, C. Jefferson, B. Martinez Hernandez, I. Miguel. Essence: A Constraint Language for Specifying Combinatorial Problems. *Constraints* 13(3), 268-306, 2008.
10. I. Gent, W. Harvey, T. Kelsey, S. Linton. Generic SBDD Using Computational Group Theory. CP, 333-347, 2003.
11. G. Katsirelos, F. Bacchus. Generalized NoGoods in CSPs. AAAI, 390-396, 2005.
12. A. K. Mackworth. Consistency in networks of relations, *Artificial Intelligence* 8, 99-118, 1977.

13. P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Comp. Intelligence* 9, 268-299, 1993.
14. P. W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence* 21, 117-133, 1983.
15. C. Schulte, P. J. Stuckey. Speeding up constraint propagation. CP, 619-633, 2004.
16. P. J. Stuckey, M. J. Garca de la Banda, M. J. Maher, K. Marriott, J. K. Slaney, Z. Somogyi, M. Wallace, T. Walsh. The G12 Project: Mapping Solver Independent Models to Efficient Solutions, CP, 13-16, 2005.
17. M. Wallace. Practical Applications of Constraint Programming. *Constraints* 1(1/2), 139-168, 1996.