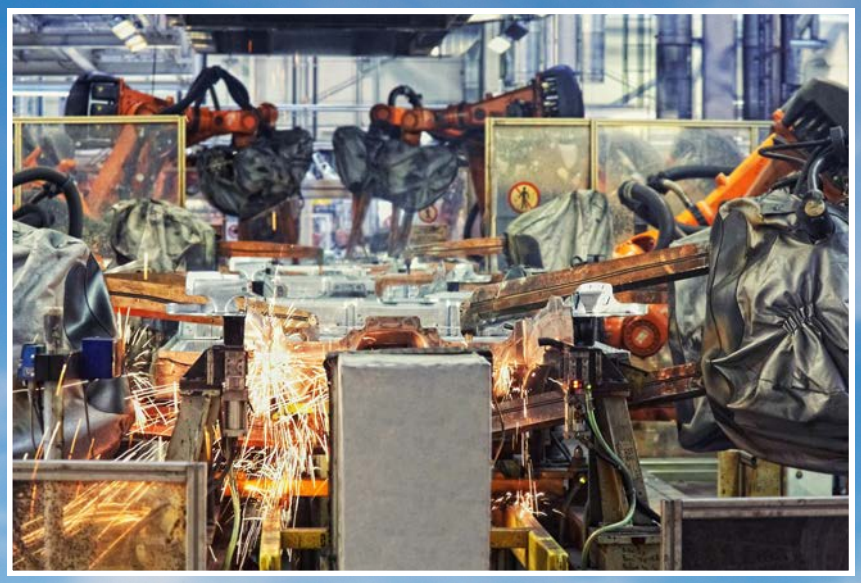
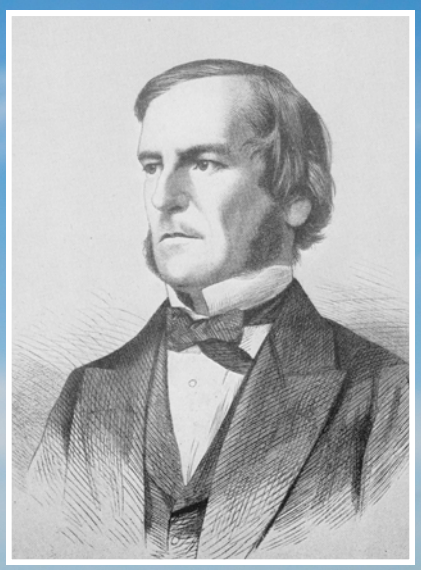
Problem Solving for the 21st Century

Efficient Solvers for Satisfiability Modulo Theories

A Knowledge Transfer Report from the London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering

By Clark Barrett, Daniel Kroening and Tom Melham





Copyright © 2014 by Clark Barrett, Daniel Kroening and Tom Melham

Front cover image credits:

Top left: Nataliya Hora / Shutterstock.com
Top right: Mile Atanasov / Shutterstock.com
Bottom left: G Tipene / Shutterstock.com
Background: Serg64 / Shutterstock.com

PROBLEM SOLVING FOR THE 21ST CENTURY

Efficient Solvers for Satisfiability Modulo Theories

By Clark Barrett, Daniel Kroening and Tom Melham

Contents

June 2014

A Knowledge Transfer Report from the London Mathematical Society and
the Smith Institute for Industrial Mathematics and System Engineering
Edited by Robert Leese and Tom Melham

London Mathematical Society, De Morgan House, 57–58 Russell Square, London WC1B 4HS
Smith Institute, Surrey Technology Centre, Surrey Research Park, Guildford GU2 7YG

AUTHORS

Clark Barrett is Associate Professor of Computer Science at New York University. He received his PhD from Stanford University in 2003, advised by David Dill. His PhD thesis laid the groundwork for the architecture of modern Satisfiability Modulo Theories (SMT) solvers. He has written numerous articles on SMT and has built several SMT systems, the latest of which is CVC4. Professor Barrett was a founder of the SMT competition, and helps lead the SMT-LIB initiative, an effort to create standards and develop resources for the SMT community. He received the IBM Software Quality Innovation Award in 2008 for the CVC3 SMT solver, and he was a recipient of the Haifa Verification Conference award in 2010 for his pioneering work in SMT.

cs.nyu.edu/~barrett/ barrett@cs.nyu.edu

Daniel Kroening is Professor of Computer Science at the University of Oxford and a Fellow of Magdalen College. He has received the M.E. and doctoral degrees in computer science from the University of Saarland in 1999 and 2001, respectively. He joined the Model Checking group in the Computer Science Department at Carnegie Mellon University in 2001 as a Postdoctoral Researcher, and was assistant professor at the Swiss Technical Institute in Zurich from 2004 to 2007. His research interests include automated formal verification of hardware and software systems, program analysis, automated testing, decision procedures, embedded systems, and hardware/software co-design.

www.kroening.com kroening@cs.ox.ac.uk

Tom Melham is Professor of Computer Science at the University of Oxford and a Fellow of Balliol College, where he is Tutor in Computation. He is also Associate Head (Research) of Oxford's Mathematics, Physical and Life Sciences Division. Melham received his PhD from the University of Cambridge in 1990 for his foundational research in mechanised reasoning. He was appointed to a Professorship of Computing Science at Glasgow in 1998, before moving to Oxford in 2002. Melham's research expertise includes deductive theorem proving, software architectures for formal reasoning tools, verification of software, firmware and hardware, formal modelling of systems, combined model checking and theorem proving, abstraction techniques, and integrating formal verification into hardware design methodologies. He works closely with leading companies in microelectronics design on advanced tools and methods for chip design validation.

www.cs.ox.ac.uk/tom.melham/ melham@cs.ox.ac.uk

Executive Summary

What is the best way to allocate assets across an investment portfolio to minimise risk? How should an airline, operating on razor-thin profit margins, assign flight crew to flights to minimise costs—at the same time meeting regulations and ensuring the schedule is robust? What is the most effective way to test a software system in a limited time? Are there any unforseen security holes in a new business-critical computer system?

All these practical problems involve finding solutions to complex systems of constraining requirements that can be formulated mathematically. The task resembles problem-solving in school maths: formulate some equations that relate quantities in the problem to be solved, and then find the right values for the variables that make the equations true. In business and industry, however, the problems are vastly larger and the mathematics much more complex and varied.

These important problems cannot be solved by hand, but must be tackled by computer software algorithms. A prominent example is *linear programming*, a mathematical optimisation technique with wide applications in modern company management and microeconomics. First used in earnest for planning in World War II, linear programming has been a mainstay of business and industry since the 1950s.

Enter SMT

Over the past decade, a new and revolutionary problem-solving technology has emerged: *Satisfiability Modulo Theories*, or 'SMT' for short. Like linear programming, it is a computerised method for finding solutions to business and industrial problems expressed mathematically by systems of constraints. But SMT can handle a richer language of constraints than linear programming, and the method encompasses a more varied range of mathematical concepts—so it has the flexibility to tackle many different kinds of problems.

With established success in the engineering design of computer chips, software that implements SMT does have limits to the size of problem it can handle—but it has also seen truly astonishing increases in speed and capacity over the past decade.

The core SMT algorithms are generic and not special to a particular problem. So, end-users who can frame their practical business and industrial problems in a mathematical way suitable for SMT automatically benefit from intense investment by the highly skilled technical specialists who develop SMT algorithms, a smart way to tap into a sophisticated technology that is improving by leaps and bounds every year.

To exploit SMT effectively, you have to express the problem to be solved in the right mathematical way. Some types of problems have well-understood translations into SMT, so the technology is ready for early adoption by at least some enterprises seeking competitive advantage. SMT solutions to other kinds of problems are the subject of active academic and industrial research—and many more lie awaiting creative discovery.

This report explains the background to SMT technology and presents several success stories. Our aim is to give a sense of the potential of SMT as an effective solution to some of today's problems—and a unique emerging technology to watch in the future.

Problem Solving with Satisfiability Modulo Theories

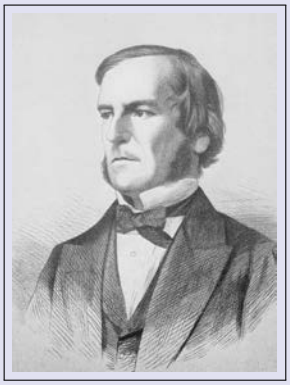
Computerised *problem solving* is the practical science of using computer algorithms to identify, from among a class of potential alternatives, a solution that meets a complex set of requirements. It begins with a real-world problem that can be formulated mathematically as a collection of conditions, or 'constraints'. A computer program then carries out mathematical calculations or reasoning to produce a solution that meets these requirements, if one exists.

Mathematical optimisation methods are an important class of problem-solving methods that seek a 'best' solution. The most prominent is *linear programming*, in which the solutions sought are usually numerical and requirements are expressed by linear constraints. Linear programming and other mainstream optimisation techniques are in daily use in business, industry, engineering, economics and management. There are dozens of commercial and open-source software packages for optimisation, including specialised mathematical modelling tools, as well as the 'solvers' themselves.

This report introduces a potentially revolutionary, but still emerging, problem-solving technology, *Satisfiability Modulo Theories*, normally just called 'SMT' for short. With its roots in computer science and mathematical logic, SMT takes a completely different approach than conventional optimisation methods. The technology has already established its practical credentials in the extremely challenging world of digital circuit engineering, where it is used to mitigate the risk of expensive design errors. And, although commercial SMT solutions are rare, there are a number of high-performance academic software packages available for research and commercial use.

The distinctive strength of SMT partly lies in the way it can handle complex *combinations* of individual constraints. Requirements don't have to just impose some individual conditions simultaneously: '$C1$ must hold *and* $C2$ must hold *and* $C3$ must hold'. In SMT, any logical

Box 1: The SAT Problem and its History



Modern symbolic logic began in the 19th century with Boolean algebra, introduced in 1847 by the English mathematician George Boole [9]. Boolean algebra introduced the idea of expressing logical relationships by algebraic formulas. The operations of the algebra include conjunction 'AND' and disjunction 'OR', together with negation 'NOT'. Unknown truth values are named by variables. So, for example, x AND (y OR NOT z) means x is true and either y is true or z is false (or both). As the American mathematician Claude Shannon observed in the 1930s, these operations provide a basis for design and analysis of digital electronic circuits. And of course they are the fundamental mathematics of all modern digital computer technology.

Logic is of central importance in Computer Science and its applications [32] and a range of sophisticated computer data structures and algorithms have been invented to analyse formulas of Boolean algebra. One extremely important class of algorithm is 'SAT solvers'. A formula is 'satisfiable' if there is an assignment of truth values to its variables that make it true. An algorithm that solves the 'SAT problem' is one that, given any Boolean formula, can determine if it is satisfiable or not. In practice, most algorithms also deliver an actual satisfying assignment of values to the variables if the formula is indeed satisfiable. SAT was the first problem to be shown to be 'NP-complete', which implies that there is no known algorithm that efficiently determines satisfiability of every possible Boolean formula.

But even though this theoretical analysis suggests that SAT is hard, modern SAT solvers can be remarkably effective on the types of satisfiability problems that arise in real-life problems. Most of these procedures have their roots in the so-called DPLL algorithm for Boolean satisfiability, introduced in 1962 by mathematicians and computer scientists Martin Davis, Hilary Putnam, George Logemann and Donald W. Loveland. Honed over years of intense competition between research groups, today's highly engineered DPLL procedures can tackle huge formulas with millions of variables. Already the cornerstone of formal verification for chip design, modern SAT solvers have immense promise as practical algorithmic tools in many other areas.

combination of conditions can be used. For example, 'exactly one of $C1$ or $C2$ must hold, but if $C1$ holds then $C3$ must not hold'. At the heart of SMT lies an algorithm called 'SAT' that can solve logical formulas of any kind, including extremely large ones (see Box 1).

SMT algorithms also encompass a more varied range of mathematical 'subject matter' than typical optimisation solvers. Virtually all SMT solvers support linear numerical constraints—indeed, they employ some of the same fundamental algorithms for these as conventional optimising solvers. Some also allow nonlinear constraints, which are in principle much harder to solve.

But the mathematical 'vocabulary' of SMT solvers goes well beyond this numerical, arithmetic domain. In particular, it includes a wide range of the mathematical concepts that are needed to model and analyse computer systems—digital hardware and computer software, and systems built from them. This modern engineering domain is completely outside the scope of traditional optimising solvers, and is the application area in which SMT solvers currently excel. But, because SMT can handle arbitrary logical complexity and encompass a variety of mathematics, it has unique potential to tackle other practical problems too.

In this report, we present some success stories in the practical use of SMT. We focus on giving the reader a feel for what today's efficient SMT solvers can do. We also give just a little of the key technical background to SMT technology.

What SMT does

SMT stands for *Satisfiability Modulo Theories*. The notion of satisfiability comes from mathematical logic. An equation or formula is *satisfiable* if, by choosing appropriate values for the variables, it can be made true. For example, the equation $x^2 - 4 = 0$ is satisfiable and a solution is $x = 2$.

But what about the equation $x^2 + 4 = 0$? It depends on what *values* x is allowed to range over. This is where *theories* come in. A theory defines what values a variable can have and what the symbols in the formula mean. Assuming our example uses the theory of *real arithmetic*, x must be a real number and thus the second equation is not satisfiable.

The power of SMT comes from its ability to handle many different kinds of theories. In addition to arithmetic, SMT solvers can reason automatically about Boolean operators (such as AND, OR), arrays and matrices, digital circuits, character strings, and software data structures such as lists and trees. Many currently supported theories have largely been devised to support the ability of SMT solvers to reason automatically about computer programs and systems. But the SMT framework is *extensible*, meaning that new theories can be defined and added to suit specific application domains. Another strength of SMT is the ability to *combine* theories in any arbitrary way. SMT solvers can reason about, for example, a matrix of integers, a list of circuits, or an array of strings.

To make use of an SMT solver to solve a problem, the statement of the problem must first be *encoded* as a logical formula. SMT solvers answer questions of the form, 'Given some conditions C, is it possible for X to happen, and if so, how?' Each condition in C and X must be modelled as a logical formula called an *assertion*. The SMT solver then checks whether it is possible for all of the conditions to be true simultaneously. If so, it can give a specific solution showing how (see Box 2 for an example). SMT solvers can automatically handle formulas that include hundreds of variables, thousands of equations, and may fill several gigabytes of disk space.

Success Stories

The first major success for SMT was in flushing out design errors in the logical functioning of modern digital electronic chips. With the cost of making the first prototype of a chip running into the millions, getting it wrong can be a very expensive mistake. SMT and SAT solvers are in everyday use by chip designers as an aid to assuring the quality of their designs.

Ultimately, chips are fabricated using a collection of photolithographic 'masks', which define the microscopic structures to be laid out on the silicon chip. The cost of each set of masks is high and, if an error in logical functioning is discovered in a fabricated chip, tracking down the cause is very difficult. To have the best chance of getting it right first time, massive amounts of computer simulation are used to check the design before it is fabricated.

But such is the complexity of today's microchips that it's not even remotely feasible to simulate all their possible operations and configurations. Checking them all would take an unimaginably long time. By modelling a chip design mathematically, however, its functioning can often be expressed quite compactly by mathematical formulas. Mathematical proofs can then be used to analyse its correctness.

This is where SAT solvers and SMT come in, to do the actual proofs, which can be very large. One method, known as 'bounded model checking', checks the functionality of a chip design up to a limited number of cycles of operation. Although this method will miss any error that manifests itself after this limit, it can still often cover much more of the chip's functionality than simulation and so provides a valuable net to trap design bugs.

Box 2: An SMT Example

Consider the following scheduling problem, adapted from [24]. Suppose two employees on duty are always needed to run a certain warehouse. The warehouse operates 24/7 and each day is divided into three shifts: day, afternoon, and night. There are nine employees available to cover the shifts, and we want to come up with a schedule for these employees. We can model this as an SMT problem as follows.

Let S be a 9×7 matrix of integers so that $S(i, j)$ represents which shift the i-th employee is working on the j-th day of the week. We encode each employee's shifts as follows: day = 1, afternoon = 10, night = 100, not working = 0.

To represent the fact that every entry in S has to take one of these values, we write 63 separate constraints:

$$\begin{aligned} &S(1,1) = 0 \text{ OR } S(1,1) = 1 \text{ OR } S(1,1) = 10 \text{ OR } S(1,1) = 100 \\ &S(1,2) = 0 \text{ OR } S(1,2) = 1 \text{ OR } S(1,2) = 10 \text{ OR } S(1,2) = 100 \\ &\quad\vdots \\ &S(9,7) = 0 \text{ OR } S(9,7) = 1 \text{ OR } S(9,7) = 10 \text{ OR } S(9,7) = 100. \end{aligned}$$

Using the shift encoding we have chosen, we can easily represent the fact that we need two workers for every shift by imposing seven summation constraints:

$$\begin{aligned} &S(1,1) + S(2,1) + S(3,1) + S(4,1) + S(5,1) + S(6,1) + S(7,1) + S(8,1) + S(9,1) = 222 \\ &S(1,2) + S(2,2) + S(3,2) + S(4,2) + S(5,2) + S(6,2) + S(7,2) + S(8,2) + S(9,2) = 222 \\ &\quad\vdots \\ &S(1,7) + S(2,7) + S(3,7) + S(4,7) + S(5,7) + S(6,7) + S(7,7) + S(8,7) + S(9,7) = 222. \end{aligned}$$

This works because the only way to add nine shift numbers together to get 222 is if two of them are 100, two are 10, two are 1, and the rest are 0.

A modern SMT solver can find a solution to all these constraints—producing a viable employee schedule to run the warehouse—in less than a second. Additional constraints can easily be added to represent minimum and maximum shift lengths, minimum and maximum number of days worked in a row, and illegal shift sequences (e.g. no one should work a day shift immediately after a night shift).

In bounded model checking, the operation of the chip over a bounded period of time is represented by a formula of logic, suitable for analysis by a SAT solver or SMT. The variables in the formula represent the individual states the device has been in. A second formula is created that characterizes the occurrence of an error in one or more of these states. The conjunction of these two formulas describes a fault happening during the period covered. If this formula is satisfiable—and this can be checked by SAT or SMT—a bug can occur and the chip design is faulty.

The use of SMT for testing chip designs is a well-establised commercial reality. In the success stories that follow, we highlight some other technical areas in which SMT technology really shines. Some of these are mature applications of SMT and ready for early commercial or industrial adopters. Others are on the horizon.

Software quality

SMT is at the core of numerous methods to improve the quality of software. Automatic quality assurance for software is a very broad field, and we refer the interested reader to a survey [23]. In what follows, we highlight techniques in which SMT plays a particularly important role.

Model-based engineering for embedded systems. An 'embedded system' is a piece of technology that has a built-in computer. The computer is often invisible to the user, as in a modern car or a hearing aid. The design of a product with an embedded computing system is a highly specialised and nontrivial task, and takes years to complete. Because of time-to-market pressures, engineers typically have only very little time to test the final product. Much of the testing and design validation thus happens with the help of simulation of the design in a computer. The simulation uses a *model* of the design, that is, a description of the behaviours of the product in its environment. Combinations of nontrivial software and environment models are sometimes referred to as cyber-physical systems. Numerous design tools have been deployed to support the design of such systems. Well-known examples include Mathworks' Simulink and Wolfram's SystemModeler.

The goal of simulation is to exercise the design to learn about the final product. This can enable the engineer to identify critical load-bearing parts, and optimise them to save weight or energy. Simulation requires a *stimulus*, that is, suitable actions of the environment that bring the design to its limits. Inadequate stimuli can give the engineer a false sense of security, and may result in mis-optimisation of the design.

The validation of cyber-physical systems offers a genuine opportunity for SMT solvers. The key challenge is that environment and system dynamics are often nonlinear, which means that traditional linear constraint solvers reach their limits. Furthermore, the software that is an integral part of the design exhibits nontrivial discrete behaviours, which are also very difficult to model with linear arithmetic. By contrast, SMT solvers are exceptionally good at modelling discrete systems and are also able to reason about nonlinear dynamics at the same time.

Model checking with SMT. *Model checking* is a formal approach that can be used on models of the kind mentioned above. Model checking explores all possible states of a model (either explicitly or implicitly) and checks each state to ensure that it satisfies the required model properties. Common existing approaches to model checking are based on propositional reasoning and work extremely well on small models but have difficulty scaling to larger models. In addition, they only apply to finite-state models.

SMT solvers are being used to power a new generation of model checkers, which scale better and can handle infinite-state models. One technique leveraged by SMT solvers to accomplish this is k-*induction*, which is a generalisation of the standard mathematical (1)-induction technique. In the standard technique, in order to prove a property P holds for all times t, we try to prove $P(0)$ and the implication $P(k)$ implies $P(k+1)$. If we can prove both, it follows that $P(n)$ is true for all n. Sometimes, the property P may be true for all n, but the implication $P(k)$ implies $P(k+1)$ does not hold. In this case, we say that P is invariant but not inductive. The k-induction technique can often work for noninductive properties P. The idea is to pick some finite k and then prove that P holds for $P(0) \dots P(k-1)$, establishing that P holds for the first k time steps. The inductive step is to prove that $P(n) \dots P(n+k-1)$ implies $P(n+k)$. In other words, instead of assuming P holds in just one previous time step, we assume that it holds in the k previous time steps, a much stronger assumption.

A number of SMT-based model checkers are being developed, with very promising performance on industrial models. These include CBMC [22], the SAL infinite-state model checker developed by SRI, and the Kind [31] and PKind [35] model checkers developed at the

Figure 1: Model-based engineering helps ensure the quality of safety-critical embedded software in cyber-physical systems such as aircraft and automobiles.

University of Iowa. A very well-known application of Model Checking to software is Mirosoft's Driver Verifier project [4]. Its current version entirely relies on SMT solving to increase the reliability of Windows device drivers. Windows device drivers are difficult to implement, owing to a plethora of programming rules. The Windows Driver Verifier ships with specifications for a good subset of these rules, and generates a report for the driver developer identifying which rules are implemented correctly. The driver verifier relies on guessing *invariants* for the loops. A loop invariant is a property that holds at the beginning of the loop. If the guess is right, correctness of the driver can be shown by means of mathematical induction.

Formal program correctness. For software that requires the highest levels of assurance, no expense may be too high. If the engineer writes the necessary invariants together with the software, SMT solvers can be used to automatically check that these invariants ensure an airtight case for the robustness of the software.

As an example, the UK air traffic control system uses software for predicting the trajectory of aircraft and for conflict detection. These features are implemented using over 200 KLOC of ADA source code by engineers at Altran in Bath. A tool then analyses the source code, and automatically generates 120,000 formulas that prove exception-freedom of the software.

Generating tests for better software

Software in cars and aircraft. Safety-critical embedded software, e.g. in cars or aircraft, has to satisfy stringent safety standards.

The goal is to root out dangerous software bugs through systematic testing. But how do we know we have done enough testing? The completeness of the testing process is frequently measured using a *coverage metric*. Well-known coverage metrics include *location coverage*, *branch coverage* and *decision coverage*. There are also combinations of these, e.g. the safety standard DO-178B requires that software for aircraft is tested with *modified condition/decision coverage* (MC/DC). There are similar standards for cars and trains.

To compute the coverage metric we first determine the *test goals*. A test goal is a reachability property, e.g. 'the execution has reached line 42' could be a goal for location coverage. A good-quality test suite is a set of input sequences that drive the system into states that cover a large percentage of those goals. The job of creating of a good test suite is still largely manual, and frankly, engineers hate this kind of work. How can we use SMT to make the computer do this for us?

The basic idea is simple: we take the program and translate it into a formula C. Every solution to the formula C computed by our solver corresponds exactly to one run of the program. Suppose we have got test goals G_1, G_2, G_3 and so on

to G_n. We then pass the formula

$$C \text{ AND } (G_1 \text{ OR } \ldots \text{ OR } G_n)$$

to the solver. This means we want an execution of the program that covers at least one (but hopefully many) of our test goals. We will then look at the solution given by the solver. Most importantly, the solution contains the program inputs that have to be fed into the program, and we will add those inputs to our test suite. We will then identify which goals are covered by the corresponding program execution. The goals covered can be scratched out of our set of goals, and we will repeat this process until we either cover all goals, or the solver tells us that the formula has no solution. In the latter case, this means that the remaining goals cannot be covered by any execution.

Similar ideas are implemented in a range of commercial tools for testing safety-critical embedded software, e.g. the Embedded Code Analyzer by TCS and the BTC-ES Embedded Tester.

The CBMC tool is an academic implementation of this technique, maintained at the University of Oxford. CBMC is a *Bounded Model Checker*, that is, the executions of the program are explored up to a user-provided depth [15]. The key benefits of this approach are that it is highly accurate (no false alarms are generated) and that it is able to generate counterexamples that aid debugging and serve as test vectors. The key disadvantage is that behaviours that require longer executions to manifest are missed by the tool. The CBMC tool won Gold in the 2014 software verification competition.

SMT for testing large-scale software. Not all software is safety-critical. Buggy software programs are an annoyance, nevertheless, and in the best case simply waste countless hours of their users' time. Getting the bugs out of these programs by inspecting their source code or by simply testing them is very costly as well, and testing and debugging is already the most time-consuming and most expensive task in software engineering. More automation of testing and debugging is thus urgently needed.

We have already explained how to use SMT for generating test inputs for embedded software. The key idea for that was to turn the entire program into a formula, which is then fed to the solver. However, programs in many application areas are simply too large for that, with millions of lines of code. It is clear that we will need to find a way to split up the problem into smaller pieces.

One way to make good use of our solver in testing is as follows. Suppose we begin with an arbitrary test input to our program. The test input will exercise our program in one particular way. We will record the behaviour of the program by noting down the sequence of statements that are executed by the computer. This sequence is called the *program path*, and if there is a bug, some program path will expose it. The problem is that even small programs can have billions of program paths, and it is tricky to pick the right inputs that expose the bug.

The idea of *Directed Automated Random Testing* (DART) [30] is as follows. Instead of encoding the entire program, we will pick some small subset of the instructions that are on the program path we have recorded earlier, and encode just those. The instructions that DART chooses to encode are, in general, only a tiny fraction of the full program, and we therefore obtain a much more tractable SMT formula. As we aim to exercise a different path, we will furthermore add a constraint to the formula that says that at least one of the branch decisions along the path needs to turn out differently. This way, the solver will generate new inputs for us that will trigger the execution of a new path that we have not yet explored. DART can be combined with heuristics that pick branches to explore that are particularly promising, e.g. by directing the execution towards a portion of the code that is known to be problematic.

Computer security

Automatic Exploit Generation (AEG) [2, 3] is an SMT-based technique for finding security vulnerabilities in software. The basic approach is as follows. A software program together with some basic properties that programs should have (e.g. a program should not access data from another program) are fed to an analyser. The analyser chooses a particular path through the program. It then creates an SMT formula which models running the program along that path while at the same time violating at least one of the given basic properties the program is supposed to have. If the formula is satisfiable, this means that for appropriate program inputs, it is possible to have the program take the path in question and violate one of the program properties. The required inputs are easily obtained from the satisfying assignment provided by the SMT solver.

If such an input is found, there is definitely a bug in the program, but it is still not clear whether the security of the system can be compromised. To determine this, an additional formula is constructed

Figure 2: The cryptography used in keyless entry fobs from 10 years ago may be susceptible to brute-force attacks by modern SMT solvers.

which is satisfiable if there exists a way to exploit the bug for a security attack. This formula is also sent to the SMT solver. If the second check succeeds, a security vulnerability has been found. Due to the power of SMT solving, this entire process is *automatic* and only takes a few seconds.

The Mayhem tool is a robust implementation of AEG available from ForAllSecure Inc., a company founded by the team who pioneered this work at Carnegie Mellon University. For more information, see Appendix 1.

Network security. As computer networks continue to grow and become more sophisticated, ensuring that network-accessible data is secure becomes more challenging. Of course, finding ways to *automate* this task are essential for scalability. SMT technology is being leveraged in several ways to aid in this important task.

Network protocols are required whenever two or more parties need to coordinate an exchange of information over the network. Often, the information is sensitive and needs to be exchanged securely. *Protocol manipulation attacks* are actions that can be taken by a third party to disrupt the exchange by either stealing sensitive information or misleading one of the parties into accepting false information. Discovering manipulation attacks is challening because they usually involve multiple distinct actions by third parties over time. MAX [38] is a manipulation attack detector that takes as input an implementation of a protocol and attempts to generate attacks against that implementation. To do this, it must explore many possible code behaviors. This is done by using SMT to model the question of how to get new behaviours from a piece of code. The solutions provided by the SMT solver can be translated into inputs that create the new behaviour.

Access control mechanisms are used to specify precisely who is allowed to access a piece of data. *Role-based access control* is popular because of its flexibility. In [29], a language called ΦRBAC is described which allows users to define flexible and high-level RBAC mechanisms which are then automatically translated into WebDSL, a special language for high-level web application development. An SMT solver is used to check automatically that the policies specified by the user have no contradictions and cover all cases, checks that would be very difficult to do manually.

OpenFlow [42] is a new and open standard for managing flow in enterprise networks. It allows network managers to write custom programs that can dynamically change how data flows through the network. While very flexible, OpenFlow creates new security challenges. FLOVER [55] is a system which takes as input a set of OpenFlow rules and an underlying security policy, and translates them into a set of SMT assertions. If the SMT solver finds a solution, this can be used to show how the OpenFlow rules violate the security policy. If no solution is found, this means that the rules follow the security policy.

Attacking old crypto. Got some old crypto? Say a keyless entry system for a car? If you are set out to demonstrate its weaknesses, no need to hire Bruce Schneier. In many cases, a workable exploit can be produced even by those not versed in cryptoanalysis. SMT, or more specifically the bit-vector theory, is typically good enough to do the job.

Breaking a previous-generation crypto implementation is often easily done as follows. Suppose that a system uses a (supposed) one-way hash-function h for authentication purposes. The most important property of such a hash-function is that given the

value $h(x)$ for some x, it is difficult to obtain a so-called collision, i.e. some y that is not x but nevertheless yields the same hash. Using SAT/SMT, it often suffices to simply give the formula

$$\exists y.\, h(y) = h(x) \wedge x \neq y$$

to your favourite solver. This was already demonstrated in 2006 for the well-known cryptographic hash functions MD4 and MD5 by researchers at Microsoft [44]. The good news, at least for those among us concerned about security of modern-day systems, is that cryptography in current systems is beyond the reach of SAT and SMT.

Scheduling and optimisation

Scheduling refers to the 'optimal allocation of scarce resources to activities over time' [36]. Examples include the scheduling done by a computer operating system in order to run multiple programs simultaneously, dividing tasks among a group of workers while satisfying certain constraints, and organising the sequence of events that must occur in order to complete a product in a factory assembly line.

A variety of techniques exist for solving scheduling problems. Recently, a number of studies have looked at using SMT techniques for scheduling. The example from Box 2 illustrates an example adapted from one of these studies on workforce scheduling [24]. In that study, 20 workforce scheduling problems were encoded as SMT problems. Two alternative encodings were investigated, one using arithmetic and the other using bit-vectors. These approaches were tried on a number of solvers and results were compared with those obtained using state-of-the-art scheduling tools. The SMT bit-vector approach performed the best of all *exact* solvers (meaning solvers that search all possibilities rather than just heuristically searching some subset of possibilities), solving 18 of the 20 problems.

Often, scheduling problems include an optimisation condition: what is desired is not just any schedule satisfying the constraints, but a schedule which, in addition to satisfying the constraints, also minimises some cost function (maximising can be handled analogously). Fortunately, SMT can be extended to handle optimisation by adapting classical techniques such as branch-and-bound and binary search [14, 45]. Experimental results using SMT for optimisation are promising. In [45], the authors show that branch-and-bound can be used effectively to solve *CELAR Radio Link Frequency Assignment problems*. These problems are not easily solved by traditional integer linear programming techniques, and the best-known approaches use Constraint Satisfaction Programming (CSP). The SMT technique outperforms the best CSP tools on the most difficult instances. Another study [1] looks at using SMT techniques to solve the *Resource-Constrained Project Scheduling Problem*. The best SMT techniques use a mix of

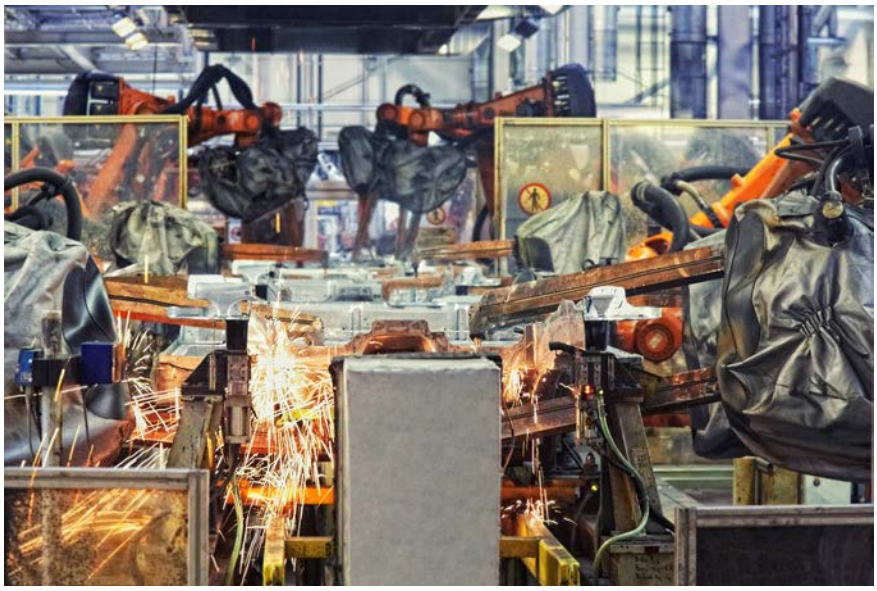


Figure 3: Determining the most efficient sequence of events in a factory assembly line is a challenging scheduling problem.

branch-and-bound and binary-search. Together, they solve more problems overall than the best integer linear programming techniques (a hybrid SAT/CSP technique is also shown to perform well). A final study [49] again shows that SMT techniques are competitive with linear programming techniques, doing especially well on *strip-packing* problems. This final study also highlights an additional important point. Linear programming solvers typically use floating point arithmetic and are thus subject to rounding errors. As a result, they may occasionally give incorrect results regarding the satisfiability of a set of constraints. SMT solvers use exact multi-precision arithmetic and thus do not have this weakness.

In 2010, the authors of the first study mentioned above founded a company, Barcelogic, which specialises in SAT and SMT-based scheduling and optimisation. For more information, see Appendix 1.

SMT in Depth

As mentioned above, SMT solvers answer questions of the form, 'Given some conditions C, is it possible for X to happen, and if so, how?' In this section, we take a closer look at how to use an SMT solver to answer such questions. The key is to understand the *language* in which conditions are expressed and various techniques for encoding real-world problems into this language.

The language of SMT

At the most abstract level, the language of SMT is the language of mathematical logic. This means that it includes things that are familiar from high school mathematics, such as arithmetic, functions, algebra, logical reasoning, etc. But it is more powerful than any of these simple components. We will introduce the language using examples.

Arithmetic. Consider the following problem: *Alice is twice as old as Bob was three years ago and Bob is three times as old as Alice was four years ago. How old are Alice and Bob?* This problem can be modelled using simple arithmetic operations. If a is Alice's age and b is Bob's age, then the situation is described by two equations: $a = 2(b-3)$ and $b = 3(a-4)$. In the terminology of SMT, we would call each equation an *assertion*. Below, we show an interactive session using the CVC4 SMT solver to solve this problem.

```
>cvc4 --dump-models --int
CVC4> a,b:INT;
CVC4> ASSERT a = 2*(b - 3);
CVC4> ASSERT b = 3*(a - 4);
CVC4> CHECKSAT;
sat
a : INT = 6;
b : INT = 6;
```

First, we tell the solver that we will be using two variables, `a` and `b`, both of which are `INT` (integers). Next, the two assertions are given. Then, we issue the `CHECKSAT` command, which tells the solver to check the assertions it has seen so far. The solver replies either `sat` (for satisfiable), meaning there is a solution, or `unsat` (for unsatisfiable), meaning there is no solution. In this case, there is a solution, and so the solver can then provide a *model*, which simply means a value for each variable. These assertions have a unique model, `a = 6` and `b = 6`. If there is more than one model, the solver will return a random one. In such cases, we can rerun the solver with an additional assertion ruling out the previous model. By repeating this, we can find all possible models.

The language of SMT includes equations, inequalities, and basic arithmetic operations like addition, subtraction, multiplication and division. Variables can range over integers or real numbers. The assertions shown above actually fall into a simple arithmetic fragment called *linear* arithmetic (linear arithmetic does not allow multiplication unless one of the two terms being multiplied is a numeric constant). Efficient algorithms exist for linear arithmetic. Nonlinear arithmetic is much more difficult (especially over integers) and is not supported by all solvers.

This highlights one challenge for users of SMT solvers: seemingly small differences in the language being used can lead to huge differences in what the solver can solve. This is part of a larger and well-studied area of computer science called computational complexity.

Boolean reasoning. Consider the following problem: *Alice is either twice as old as Bob or 2 years younger than Bob. Bob is either twice as old as Alice or 3 years younger than Alice. Alice's age is more than 0 and is not 2.* The SMT input for this problem is shown below.

```
> cvc4 --int --dump-models
CVC4> a,b:INT;
CVC4> ASSERT a=2*b OR a=b-2;
CVC4> ASSERT b=2*a OR b=a-3;
CVC4> ASSERT a>0 AND NOT a=2;
CVC4> CHECKSAT;
sat
a : INT = 6;
b : INT = 3;
```

Notice the use of `OR`, `NOT`, and `AND`. These are called *Boolean operators*. SMT allows sophisticated assertions to be built up using any combination of simple assertions and Boolean operations. The ability of SMT to reason efficiently even when given arbitrary Boolean expressions is one of its important strengths. Note that, by contrast, linear programming tools are typically not very effective for constraints that use `OR` and are thus more limited in how easily they can be applied to certain real-world problems.

Functions. A surprisingly useful construct supported by SMT is the notion of an *uninterpreted* function, meaning a function about which we know nothing other than that it is a function. These functions can be used for modelling in a variety of ways. Consider the following simple example. Suppose we want to show automatically that if $a = b$ then $ax = bx$. In order to avoid having to reason about nonlinear

arithmetic, we can instead use *abstraction* to replace multiplication by an uninterpreted function, m, and instead try to show that if $a = b$ then $m(a, x) = m(b, x)$. Showing this only requires reasoning about addition and functions. Note that because multiplication is a function and m is definitely some function, if we can show the abstract property, this also proves that the original property holds. We can show the abstract property by showing that that it is impossible for $a = b$ to be true while $m(a, x) = m(b, x)$ is false.

```
> cvc4 --int --dump-models
CVC4> a,b,x : INT;
CVC4> m : (INT,INT) -> INT;
CVC4> ASSERT a = b;
CVC4> ASSERT
  NOT m(a,x) = m(b,x);
CVC4> CHECKSAT;
unsat
```

Bit-vectors and arrays. One of the most successful applications of SMT has been in the analysis of hardware and software. For these analyses, the ability to reason automatically about computer representations of data is crucial. The bit-vector theory provides this capability. A bit-vector is a fixed-length string of 0's and 1's. SMT solvers can reason about bit-vectors and many interesting operations on them including concatenation and extraction, bit-vector arithmetic, and bit-wise Boolean operations. Consider the following example, which tries to find nonzero bit-vectors `a` and `b` (of length 2 and 4 respectively) such that concatenating `a` with `b` gives the same result as concatenating `b` with `a`. Concatenation is represented by the `@` symbol, and binary constants are represented as strings of `0`'s and `1`'s prefixed with `0bin`.

```
> cvc4 --dump-models --int
CVC4> a : BITVECTOR(2);
CVC4> b : BITVECTOR(4);
CVC4> ASSERT a @ b = b @ a;
CVC4> ASSERT NOT (a = 0bin00);
CVC4> CHECKSAT;
sat
a : BITVECTOR(2) = 0bin01;
b : BITVECTOR(4) = 0bin0101;
```

Another crucial capability for reasoning about hardware and software is the ability to model memory. This is provided by the theory of arrays. Arrays are indexed collections of elements. The language of arrays includes an operation to read an array at some index and an operation to write a new value to an array at some index. Reading an array `a` at index `i` is written `a[i]`, and writing a new value `x` at index `i` is written `a WITH [i] := x`. As an example, consider trying to show that in some cases, writing to an array at index `i` and then `j` is not the same as writing to the same array at index `j` and then `i`.

```
>cvc4 --dump-models --int
CVC4> a,b1,b2,c1,c2 :
  ARRAY INT OF INT;
CVC4> i,j,x,y : INT;
CVC4> ASSERT b1 =
  a WITH [i] := x;
CVC4> ASSERT b2 =
  b1 WITH [j] := y;
CVC4> ASSERT c1 =
  a WITH [j] := y;
CVC4> ASSERT c2 =
  c1 WITH [i] := x;
CVC4> ASSERT NOT b2 = c2;
CVC4> CHECKSAT;
sat
i : INT = 0;
j : INT = 0;
x : INT = 0;
y : INT = -1;
```

The above example shows that when `i` and `j` are the same but the values written are different, the resulting arrays are different as well.

Quantifiers. Quantifiers are a powerful logical construct that allow one to generalise statements. For example, asserting $f(0) = 0$ states that the value of f at zero is zero. The *existential* quantifier $\exists$ can be used to generalise this by saying that f is zero at some point, but we don't know which. This is expressed as $\exists\, x.\ f(x) = 0$. On the other hand, the *universal* quantifier $\forall$ can be used to generalise by saying that f is zero at all points. This is written $\forall\, x.\ f(x) = 0$. The following example shows how this last fact can be used to show that it is not possible for there to be an integer y such that $f(y) = 1$ (in the CVC4 language, universal quantification is expressed with `FORALL`).

```
> cvc4 --dump-models --int
CVC4> f : INT -> INT;
CVC4> y : INT;
CVC4> ASSERT
  FORALL (x:INT): f(x) = 0;
CVC4> ASSERT f(y) = 1;
CVC4> CHECKSAT;
unsat
```

While some SMT solvers do support the use of quantifiers, using quantifiers can make the problem much harder. In general, the use of quantifiers increases the computational complexity of the problem. Even SMT solvers that do support quantifiers will fail to solve some problems that use quantifiers, either by running for a long time with no answer, or by returning the answer `unknown`.

Putting it all together: a software security example

Consider the following simple piece of C code:

```
int getPassword()
```

```
{
char buf[4];
gets(buf);
return strcmp(buf,"SMT");
}

void main()
{
int x = getPassword();
if (x) {
printf("Access Denied\n");
exit(0);
}
printf("Access Granted\n");
}
```

When this program is run, it waits for an input from the user. If the user enters 'SMT', then they get the 'Access Granted' message. Otherwise, they get the 'Access Denied' message. Or at least, that is what is supposed to happen. It turns out that this program has a security vulnerability—a *buffer overflow* attack can be used to get the 'Access Granted' message even without knowing what the correct input is (on a Linux x64 platform running gcc 4.8.2, an input consisting of 24 arbitrary characters followed by `]`, ⟨*ctrl-f*⟩, and `@`, will bypass the 'Access Denied' message).

Briefly, the problem is that when the `gets` function is called, there is no limit to the size of the input that the user can provide. However, the program only reserves enough space for an input of size 4. If the user provides a longer input, then those input characters will run over into other parts of the computer memory including a part of memory that stores the location of the code to be run after `getPassword` completes. With the right input, we can trick the program into jumping to the line of code that prints "Access Granted".

But what is the right input? We can model the problem using SMT. We will use bit-vectors to model the internal state (called *registers*) of the CPU while executing the program, and we will use arrays of bit-vectors to model the computer memory and the user's input.

When `getPassword` is called, the location of the code to run when `getPassword` completes (the `if` statement) is put into a special location in memory called the *stack*. The top of the stack is pointed to by the internal *stack pointer* register (`sp` for short). When `getPassword` is called, the top of the stack contains the address of the `if` statement. The first thing that happens after `getPassword` is called is that `sp` is decreased by 4 to make room for the 4-character array `buf` (which is also stored on the stack). Next, the call to `gets` writes input characters into memory starting with the location pointed to by `sp`. When `getPassword` is finished, `sp` is increased by 4, reclaiming the memory occupied by `buf`, and then the *instruction pointer* register (`ip` for short) is loaded with the location pointed to by `sp`. We wish to determine whether it is possible to set `ip` to a value that we choose instead of the location of the `if` statement. The SMT formula shown in Figure 4 models a simplified version of the above program.

We use a couple of encoding tricks here. First is something called static single assignment (SSA). SSA represents the same variable at different times by using different names for the different times. For example, the stack pointer is represented initially by `sp0`, and then later by `sp1`, and then finally by `sp2`. As the variable changes, the relationship between its previous value and new value is captured with a formula that relates them. For example, the expression `sp1 = BVSUB(8,sp0,0bin100)` tells us that the new value of `sp` is equal to the old value minus 4 (the first parameter of 8 is used to specify the bit-width of the operation).

Another trick is *loop unrolling*. We have modelled the call to `gets` as a series of 5 writes to memory. This corresponds to an input of 5 characters, meaning that a loop that reads a single character would get executed 5 times. How do we know how many times to unroll the loop? In general, we don't, but in practice we can start small and try more and more loop unrollings until we find a satisfying assignment or give up. For this example, if we use 4 or fewer unrollings, the solver returns `unsat`. However, with 5 characters, the result is `sat`. In particular, with an input of 5 characters, the last character of the input will become the new value of `ip`.

Several other simplifications are made in this example for ease of understanding and exposition: we use a memory word size of 8 bits (1 byte), we don't model the final `NULL` character of the input, and we ignore the base pointer (`bp`). However, these details can easily be added. Encodings like the one sketched here can be used to automatically detect bugs and vulnerabilities in sophisticated software programs.

The SMT-LIB Initiative

The SMT-LIB initiative is a broad effort whose goal is to provide resources and direction for both SMT developers and users. One of its main goals has been to produce and maintain a language standard for SMT inputs, called the SMT-LIB language, and encourage SMT developers to support the language. Version 1 of the language standard became

```
sp0,sp1,sp2:BITVECTOR(8);
ip:BITVECTOR(8);
m0,m1,m2,m3,m4,m5 : ARRAY BITVECTOR(8) OF BITVECTOR(8);
in : ARRAY INT OF BITVECTOR(8);
ASSERT sp1 = BVSUB(8,sp0,0bin100);
ASSERT m1 = m0 WITH [sp1] := in[1];
ASSERT m2 = m1 WITH [BVPLUS(8,sp1,0bin1)] := in[2];
ASSERT m3 = m2 WITH [BVPLUS(8,sp1,0bin10)] := in[3];
ASSERT m4 = m3 WITH [BVPLUS(8,sp1,0bin11)] := in[4];
ASSERT m5 = m4 WITH [BVPLUS(8,sp1,0bin100)] := in[5];
ASSERT sp2 = BVPLUS(8,sp1,0bin100);
ASSERT ip = m5[sp2];
ASSERT NOT ip = m0[sp0];
CHECKSAT;
```

Figure 4: SMT formula generated from our software security example in C.

available in July 2004. Version 2.0 was released in March 2010. It is expected that additional updates will take place periodically. Note that the examples in this report are in the CVC4 language rather than the SMT-LIB language. We chose the CVC4 language because it is a little easier for humans to read, whereas the SMT-LIB language is more difficult to read because of its use of prefix notation. On the other hand, the SMT-LIB language is easier for machines to process quickly. To illustrate the difference, we show below the SMT-LIB translation of the first example from the Arithmetic section above (`QF_LIA` is an SMT-LIB logic designation for quantifier-free linear integer arithmetic).

```
(set-logic QF_LIA)
(set-info
     :smt-lib-version 2.0)
(declare-fun a () Int)
(declare-fun b () Int)
(assert (= a (* 2 (- b 3))))
(assert (= b (* 3 (- a 4))))
(check-sat)
(exit)
```

A closely related goal of the initiative has been to collect a large and representative sample of application benchmarks in the standard language. The SMT-LIB benchmark library now includes over 100,000 benchmarks covering 25 logics (a *logic*, as used in SMT-LIB, is a particular subset of the full SMT language). The benchmarks are now standard metrics used in academic papers and have also been used in the SMT competition (the competition has been held annually since 2005 with the exception of 2013 when a more extensive *SMT evaluation* was done instead). More information about the SMT community and publicly available SMT resources can be found under ‘Further reading’ in the ‘Next Steps’ section below.

Current Challenges

SMT solvers are becoming robust and versatile tools used in many applications, but there are still some challenges preventing more widespread use and adoption.

1. There are a number of mature SMT solvers that are commercially available (or free) and well supported (see Appendix 3). But SMT solvers are typically used as back ends to more application-specific software tools. This means that to use an SMT solver, you need an appropriate front end for your application. Unfortunately, there are not yet very many commercial applications available that leverage SMT technology. Most projects using SMT solvers are either research prototypes or company-internal products not available outside (some exceptions are listed in Appendix 1).

   This means that if you want to use SMT in your domain, you may have to get involved with the development of a custom front end, including determining the best way to model your problem using logical formulas. This can also be seen as an opportunity, as such a project would put you in a position to shape a domain-specific package that generates the right formulas for your particular problem domain.

   A good way to do this is to create a collaboration with an established SMT research group (see Appendix 2), perhaps by sponsoring a PhD student to work in your application domain. This provides a concrete way to get someone with SMT expertise to work on your problem—and if the student enjoys the work, you may be able to hire them when they graduate, thus obtaining an expert who is trained in the domain you care about.

2. Different solvers for different theories in SMT vary in their computational complexity. This means that the robustness of the tool depends a lot on the formula it is working on. If an application only produces formulas using real linear arithmetic, for example, the solver is likely to perform well and scale to large problem instances. If the application uses nonlinear arithmetic or quantifiers, the solver will be more fragile, working well on some problems but unable to solve others that seem to differ only slightly. Researchers are working on ways to make SMT solvers more robust across a variety of theories, and as these techniques become more mature, users will have a better understanding of how to avoid situations where the SMT solver is unlikely to succeed.

3. While the framework of modern SMT solvers allows new theories to be added and plugged in, in practice this is an activity that currently can only be undertaken by (or in consultation with) an SMT expert. Efforts are underway to make it possible for nonexperts to add theories by developing languages or APIs for adding new theories.

4. There are only a very few SMT experts in the world meaning that it can be quite difficult to find someone to lead an effort requiring SMT expertise. Sponsoring a student (mentioned above) is one way to overcome this challenge.

Next Steps

A few companies do provide SMT-based solutions to challenging problems. If you're fortunate enough to have one of these problems, you may want to contact one of these companies to find out more about their business solutions (see Appendix 1). If, on the other hand, your application does not seem to be covered by an existing business solution, you may want to consider partnering with an academic or industrial research group (see Appendix 2) to develop a new solution. Many researchers are interested in investigating and supporting promising new application areas.

Hands-on experience

Appendix 3 lists the most prominent SMT solvers available today. Many of these tools have associated web pages with documentation and tutorials, and some (such as CVC4) have active user groups with mailing lists. Browsing the web pages and working through some of these tutorials is a good way to learn about what SMT solvers can do.

The CVC4 input language is described at [18] with many examples. A tutorial on using the CVC4 C++ API can be found at [62]. Z3 is a popular SMT solver from Microsoft Research. An interactive web site with puzzles and tutorials is at [66].

Further reading

There are at least three textbooks that cover topics relevant to SMT solvers in some detail [10, 33, 39]. They are too technical for a general audience, but should be accessible to someone with a strong background in logic or computer science. In addition, there are several survey articles about SMT that provide a higher-level (but still quite technical) overview of relevant topics [6, 7, 48].

The SMT-LIB website [52] contains information about the SMT-LIB standard, including the official document describing the SMT-LIB language, a tutorial developed by David Cok, and a link to the SMT-LIB mailing list. It also contains many other useful links.

The SMT competition website, SMT-COMP, [51] contains results from the most recent SMT competition (and from older editions as well) . These results give some indication as to the relative performance of different SMT solvers.

Conferences and workshops

SMT is an active area of research. Conferences and workshops provide opportunities for users and developers of SMT to gather and discuss the latest breakthroughs and challenges. The primary forum for all SMT topics is the International Workshop on Satisfiability Modulo Theories, held each year (see [53]). The SMT workshop is typically co-located with a broader related conference such as the International Conference on Computer Aided Verification (CAV), the International Conference on Automated Deduction (CADE), the International Joint Conference on Automated Reasoning (IJCAR), or the International Conference on Theory and Applications of Satisfiability Testing (SAT).

Appendix 1: Business Solutions using SMT

AdaCore/Altran: SPARK is a programming language (a subset of the Ada language) with extensive tool support for formal verification. It has been used in a number of safety-critical software projects. The SPARK toolset is developed and maintained as a partnership between Altran [57] and AdaCore [58]. SMT solvers are used internally to do the verification.

Barcelogic: Barcelogic [5] is a company which uses SAT- and SMT-based techniques to solve industrial scheduling and optimisation problems. Their technology is based on the Barcelogic SMT solver, which was a frequent winner in the SMT Competition from 2005 to 2009.

BTC-ES: BTC-ES [11] offers tools for automating testing and verification of software for embedded systems that is designed with Simulink. Their tool BTC EmbeddedTester automatically generates a formula that encodes frequently required coverage criteria for safety standards such as ISO 26262.

ForAllSecure: ForAllSecure, Inc. [26] is a company whose mission is to 'test the world's software.' Their Mayhem tool uses an SMT back-end to automate bug-finding for binary programs, additionally checking to see which bugs can lead to security vulnerabilities. ForAllSecure was founded by David Brumley, a professor at Carnegie Mellon University.

Programming Research: Headquartered in Hersham, UK, Programming Research develops the PQRA tool suite for static analysis of C and C++ programs [47]. Their analysis uses solvers for the accurate and precise detection of defects like buffer overflows and use of uninitialised data.

SRI International: The Symbolic Analysis Laboratory (SAL) [60] is an open-source tool suite developed under the GPL license by SRI International for the symbolic analysis of systems. It includes tools for abstraction, program analysis, theorem proving and model checking. Many of these tools rely on an SMT solver. SAL includes the ICS SMT solver by default but works best with SRI's Yices SMT solver (which must be specifically selected at download time because it has a more restrictive license).

Appendix 2: Active Researchers and Practitioner Groups

SMT and its applications is a rapidly growing field, with many developers and users of the technology in the UK and around the world. The list given here is not intended to be exhaustive—but instead to give a cross section of the SMT community in the UK and beyond. Our apologies are offered to any group we left off this list.

In the UK

University of Cambridge: *Smten* is a unified language developed at Cambridge for general-purpose functional programming and SMT query orchestration [46]. It aims to simplify the production of practical SMT-based tools for computer aided verification. An open-source implementation is available in the form of a plugin to the Glasgow Haskell Compiler.

University of Edinburgh: Paul Jackson develops proof procedures for nonlinear arithmetic, and investigates their application to formal verification of hybrid systems [34]. He has also developed a tool, called Victor, that can translate proof obligations from the SPARK toolset into the SMT-LIB language.

University of Manchester: Internationally leading experts in first-order provers, Konstantin Korovin [37] and Andrei Voronkov [64], are also active researchers into algorithms and tools for SMT.

University of Oxford: Daniel Kroening [40] and Tom Melham [43] develop program and hardware analysis tools that use SMT-based reasoning engines. Daniel Kroening is a co-author of the *Decision Procedures* book [39].

Internationally

University of Bremen: Florian Lapschies is a researcher in Jan Peleska's operating and distributed systems group who has developed the SONOLAR SMT solver [56], a solver for arrays and bit-vectors, for use in automatic test case generation.

Fondazione Bruno Kessler: A number of researchers in the center for information and communication technology at FBK [13] helped lay the foundations for modern SMT work, including Alessandro Armando, Alessandro Cimatti, Alberto Griggio, and Silvio Ranise. Cimatti and Griggio are primary developers of the MathSAT SMT solver, which has won awards in several SMT competitions and is used in a number of industrial applications. MathSAT is also used as a backend in nuXmv [12], a model checker for infinite-state systems.

University of Freiburg: Jochen Hoenicke, Jürgen Christ, and Alexander Nutz have developed SMT Interpol [54], an SMT solver specialising in the computation of interpolants.

Technical University of Catalonia: The Logics and Programming group [41] includes Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio, who all work on SMT and its applications. They have developed the Barcelogic SMT solver and Nieuwenhuis, Oliveras, and Rodríguez-Carbonell are founders of the Barcelogic company.

Intel Corporation: Amit Goel, Jim Grundy, and Sava Krstić at Intel's Strategic CAD Labs have developed an open-source SMT solver called the Decision Procedure Toolkit [20]. They have also used SMT for software verification at Intel.

University of Iowa: Cesare Tinelli is an automated reasoning researcher with extensive expertise in SMT [61]. With Silvio Ranise, he founded the SMT-LIB initiative and continues to play a leading role in guiding the initiative. He and Clark Barrett are the project leaders for the development of the CVC4 SMT solver, an industrial-strength open-source SMT solver which is also a research platform. He is also the project leader for the Kind and PKind model checkers.

Johannes Kepler University: Armin Biere is the developer of Boolector, an SMT solver for bit-vectors and arrays [8]. Biere is an editor of the *Handbook of Satisfiability*, and decision procedures for SAT and SMT developed by him or under his guidance have ranked at the top of both the SAT and the SMT competitions.

University of Lugano: Natasha Sharygina leads the Formal Verification and Security Lab [27], which develops and applies SMT techniques. They are the authors of the OpenSMT SMT solver.

Microsoft Research (Redmond): Leonardo de Moura, formerly at SRI International where he was a co-author of the original Yices SMT solver, joined Microsoft Research in 2006 [19]. He, Nikolaj Bjørner and Christoph Wintersteiger develop the Z3 SMT solver, which has dominated recent SMT competitions and is the most widely used solver in SMT applications today.

INRIA: Pascal Fontaine [25] and David Déharbe [21] lead the development of the veriT SMT solver, which has strong capabilities in proof production and quantifier reasoning.

New York University: Clark Barrett leads the Analysis of Computer Systems (ACSys) group at NYU [63]. One major focus of the group is the development of theory and tools for SMT. In collaboration with the University of Iowa, they manage the development of the CVC4 SMT solver.

Paris-Sud University: Sylvain Conchon [17] is the primary developer of the Alt-Ergo SMT solver, with unique capabilities in reasoning about polymorphic data types and associative and commutative symbols.

SRI International: SRI [16] employs several SMT experts, including Natarajan Shankar, Bruno Dutertre, and Dejan Jovanović. They have developed several solvers including ICS, Simplics, and their current flagship SMT solver, Yices. Yices was one of the first high-performance SMT solvers, winning every division of the 2006 SMT competition.

Technion – Israel Institute of Technology: Ofer Strichman works on the development and application of SAT solvers and their optimisation, as well as research on decision procedures for first-order theories and SMT [59]. He is a co-author of the *Decision Procedures* book [39].

University of Trento: Roberto Sebastiani [50] works on SMT and its application to formal verification. He is part of the team developing the MathSAT SMT solver.

University of Waterloo: Vijay Ganesh's research interests include the application of automated reasoning techniques, including SMT solvers, to software engineering [28]. He is the primary author of the STP SMT solver, which won the bit-vector category of the SMT competition in 2006 (tied with SRI) and in 2010.

Appendix 3: SMT Solvers

Many different software implementations of SMT are available, including both research tools and commercially developed ones. The table below lists some of the more prominent and well-established and supported. A more comprehensive list is maintained in the Wikipedia entry for Satisfiability Modulo Theories [65].

Solver	License	Website
Alt-Ergo	CeCILL-C	alt-ergo.ocamlpro.com
Supported theories: *uninterpreted functions, linear integer and real arithmetic, nonlinear arithmetic, polymorphic arrays, bit-vectors, records, enumerated datatypes, ACsymbols, quantifiers*		
Barcelogic	Proprietary	www.lsi.upc.edu/~oliveras/bclt-main.html
Supported theories: *uninterpreted functions, integer and real difference logic, linear integer and real arithmetic*		
Boolector	GPLv3	fmv.jku.at/boolector
Supported theories: *arrays, bit-vectors*		
CVC4	BSD	cvc4.cs.nyu.edu
Supported theories: *uninterpreted functions, integer and real difference logic, linear integer and real arithmetic, arrays, bit-vectors, datatypes, strings, quantifiers*		
MathSAT	Proprietary	mathsat.fbk.eu
Supported theories: *uninterpreted functions, integer and real difference logic, linear integer and real arithmetic, arrays, bit-vectors, floating point arithmetic*		
OpenSMT	GPLv3	verify.inf.unisi.ch/opensmt.html
Supported theories: *uninterpreted functions, real and integer difference logic, linear real arithmetic, bit-vectors*		
SMTInterpol	LGPLv3	ultimate.informatik.uni-freiburg.de/smtinterpol
Supported theories: *uninterpreted functions, linear integer and real arithmetic*		
SONOLAR	Proprietary	http://www.informatik.uni-bremen.de/~florian/sonolar
Supported theories: *bit-vectors, floating-point arithmetic*		
STP	MIT	sites.google.com/site/stpfastprover
Supported theories: *arrays, bit-vectors*		
veriT	BSD	www.verit-solver.org
Supported theories: *uninterpreted functions, integer and real difference logic, quantifiers*		
Yices	Proprietary	yices.csl.sri.com
Supported theories: *uninterpreted functions, integer and real difference logic, linear integer and real arithmetic, arrays, bit-vectors*		
Z3	MSR-LA	z3.codeplex.com
Supported theories: *uninterpreted functions, integer and real difference logic, linear integer and real arithmetic, nonlinear arithmetic, arrays, bit-vectors, datatypes, floating point arithmetic, quantifiers*		

References

[1] Carlos Ansótegui, Miquel Bofill, Miquel Palahı, Josep Suy, and Mateu Villaret. Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem. In *Proceedings of SARA – Symposium on Abstraction, Reformulation and Approximation*. AAAI, 2011.

[2] T. Avgerinos, S. Cha, B. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2011.

[3] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, February 2014.

[4] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[5] Barcelogic. barcelogic.com, April 2014.

[6] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[7] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Ed Clarke, Thomas Henzinger, and Helmut Veith, editors, *Handbook of Model Checking*. Springer, 2014.

[8] Armin Biere. fmv.jku.at/biere, April 2014.

[9] George Boole. *Mathematical Analysis of Logic: Being an Essay Towards a Calculus of Deductive Reasoning*. Macmillan, Barclay and Macmillan, 1847.

[10] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer, 2007.

[11] BTC Embedded Systems AG - Homepage. www.btc-es.de, April 2014.

[12] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *Computer Aided Verification - 26th International Conference, CAV 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, Lecture Notes in Computer Science. Springer, 2014.

[13] FBK — Center for Information Technologies. ict.fbk.eu, April 2014.

[14] Alessandro Cimatti, Anders Franzn, Alberto Griggio, Roberto Sebastiani, and Cristian Stenico. Satisfiability modulo the theory of costs: Foundations and applications. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 99–113. Springer, 2010.

[15] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[16] Computer Science Laboratory. www.csl.sri.com, April 2014.

[17] Sylvain Conchon. www.lri.fr/~conchon, April 2014.

[18] CVC4's native language. cvc4.cs.nyu.edu/wiki/CVC4%27s_native_language, June 2014.

[19] Leonardo de Moura. leodemoura.github.io, April 2014.

[20] Decision Procedure Toolkit. sourceforge.net/projects/dpt, April 2014.

[21] David Déharbe. www.sites.google.com/site/deharbe, April 2014.

[22] Alastair F. Donaldson, Daniel Kroening, and Philipp Rümmer. Automatic analysis of DMA races using model checking and *k*-induction. *Formal Methods in System Design*, 39(1):83–113, 2011.

[23] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[24] Christoph Erkinger. Rotating workforce scheduling as satisfiability modulo theories. Master's thesis, Technische Universität Wien, Vienna, Austria, 2013.

[25] Pascal Fontaine. www.loria.fr/~fontaine, April 2014.

[26] ForAllSecure, Inc. forallsecure.com, April 2014.

[27] Formal Verification at University of Lugano — Formal Verification and Security Lab. verify.inf.usi.ch, April 2014.

[28] Vijay Ganesh. ece.uwaterloo.ca/~vganesh, April 2014.

[29] S. H. Ghotbi and B. Fischer. Fine-grained role- and attribute-based access control for web applications. In *Software and Data Technologies (Proceedings of ICSOFT 2012)*, pages 171–187, 2013.

[30] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Programming Language Design and Implementation (PLDI)*, pages 213–223. ACM, 2005.

[31] George Hagen and Cesare Tinelli. Scaling up the formal verification of lustre programs with SMT-based techniques. In *2008 Formal Methods in Computer-Aided Design*, pages 1–9. IEEE, November 2008.

[32] Joseph Y. Halpern, Robert Harper, Neil Immerman, Phokion G. Kolaitis, Moshe Y. Vardi, and Victor Vianu. On the unusual effectiveness of logic in computer science. *Bulletin of Symbolic Logic*, 7(2):213–236, 2001.

[33] John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[34] Paul Jackson. homepages.inf.ed.ac.uk/pbj, April 2014.

[35] Temesghen Kahsai and Cesare Tinelli. PKIND: A parallel *k*-induction based model checker. In J. Barnat and K. Heljanko, editors, *Proceedings of 10th International Workshop on Parallel and Distributed Methods in verification (Snowbird, Utah, USA)*, Electronic Proceedings in Theoretical Computer Science, 2011.

[36] David Karger, Cliff Stein, and Joel Wein. Scheduling algorithms. In Mikhail J. Atallah and Marina Blanton, editors, *Algorithms and Theory of Computation Handbook*, pages 20–1:20–34. Chapman & Hall / CRC Press, 2010.

[37] Konstantin Korovin. www.cs.man.ac.uk/~korovink, April 2014.

[38] Nupur Kothari, Ratul Mahajan, Todd Millstein, Ramesh Govindan, and Madanlal Musuvathi. Finding protocol manipulation attacks. *ACM SIGCOMM Computer Communication Review*, 41(4):26, October 2011.

[39] Daniel Kroening and Ofer Strichman. *Decision Procedures*. Springer, 2008.

[40] Daniel Kroening. www.kroening.com, April 2014.

[41] LOGPROG: Homepage. www.lsi.upc.edu/~erodri/logprog/homepage.htm, April 2014.

[42] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[43] Tom Melham. www.cs.ox.ac.uk/tom.melham/home.html, April 2014.

[44] Ilya Mironov and Lintao Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 4121 of *Lecture Notes in Computer Science*, pages 102–115. Springer, 2006.

[45] Robert Nieuwenhuis and Albert Oliveras. On SAT modulo theories and optimization problems. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 4121 of *Lecture Notes in Computer Science*, pages 156–169. Springer, 2006.

[46] University of Cambridge. Enhancing the Satisfiability Modulo Theories Experience. www.cl.cam.ac.uk/research/security/ctsrd/smten.html, April 2014.

[47] Programming Research, Ltd. programmingresearch.com, April 2014.

[48] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 3:141–224, 2007.

[49] Roberto Sebastiani and Silvia Tomasi. Optimization in SMT with $\mathcal{LA}(\mathbb{Q})$ cost functions. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 484–498. Springer, 2012.

[50] Roberto Sebastiani. disi.unitn.it/~rseba, April 2014.

[51] SMT-COMP. www.smtcomp.org, June 2014.

[52] SMT-LIB. www.smtlib.org, June 2014.

[53] SMT workshop. www.smt-workshop.org, June 2014.

[54] SMTInterpol – an Interpolating SMT Solver. ultimate.informatik.uni-freiburg.de/smtinterpol, April 2014.

[55] Sooel Son, Seungwon Shin, Vinod Yegneswaran, Phillip Porras, and Guofei Gu. Model checking invariant security properties in OpenFlow. In *2013 IEEE International Conference on Communications (ICC)*, pages 1974–1979. IEEE, June 2013.

[56] SONOLAR. www.informatik.uni-bremen.de/~florian/sonolar, April 2014.

[57] SPARK. intelligent-systems.altran.com/technologies/software-engineering/spark.html, April 2014.

[58] AdaCore: SPARK Pro. www.adacore.com/sparkpro, April 2014.

[59] Ofer Strichman. ie.technion.ac.il/Home/Users/ofers.phtml, April 2014.

[60] Symbolic Analysis Laboratory. sal.csl.sri.com, April 2014.

[61] Cesare Tinelli. homepage.cs.uiowa.edu/~tinelli, April 2014.

[62] Tutorials – CVC4. church.cims.nyu.edu/wiki/Tutorials, June 2014.

[63] New York University. The ACSys Group Home Page. cs.nyu.edu/acsys, April 2014.

[64] Andrei Voronkov. voronkov.com, April 2014.

[65] Wikipedia. Satisfiability Modulo Theories. en.wikipedia.org/wiki/Satisfiability_Modulo_Theories, April 2014.

[66] Z3 @ rise4fun from Microsoft. www.rise4fun.com/z3, June 2014.

Papers in the Series:

1. Managing Risk in the Modern World
Applications of Bayesian Networks
Norman Fenton and Martin Neil

2. The GPU Computing Revolution
From Multi-Core CPUs to Many-Core Graphics Processors
Simon McIntosh-Smith

3. Problem Solving for the 21st Century
Efficient Solvers for Satisfiability Modulo Theories
Clark Barrett, Daniel Kroening and Tom Melham

4. Quantitative Verification
Formal Guarantees for Timeliness, Reliability and Performance
Gethin Norman and David Parker

Problem Solving for the 21st Century

Efficient Solvers for Satisfiability Modulo Theories

A Knowledge Transfer Report from the London Mathematical Society and the Smith Institute for Industrial Mathematics and System Engineering

by Clark Barrett, Daniel Kroening and Tom Melham

The London Mathematical Society (LMS) is the UK's learned society for mathematics. Founded in 1865 for the promotion and extension of mathematical knowledge, the Society is concerned with all branches of mathematics and its applications. It is an independent and self-financing charity, with a membership of around 2,300 drawn from all parts of the UK and overseas. Its principal activities are the organisation of meetings and conferences, the publication of periodicals and books, the provision of financial support for mathematical activities, and the contribution to public debates on issues related to mathematics research and education. It works collaboratively with other mathematical bodies worldwide. It is the UK's adhering body to the International Mathematical Union and is a member of the Council for the Mathematical Sciences, which also comprises the Institute of Mathematics and its Applications, the Royal Statistical Society, the Operational Research Society and the Edinburgh Mathematical Society.

www.lms.ac.uk

The Smith Institute for Industrial Mathematics and System Engineering is a leader in the UK for harnessing mathematics as an engine of business innovation. Established as an independent organisation in 1997, it adopts a systems approach, connecting modelling, data, algorithms and implementation. It works with an extensive network of collaborators in industry, government and the university research base to improve products, services and processes. The Smith Institute represents the UK internationally in the field of industrial mathematics and works closely with UK Research Councils and other funding agencies to create new and effective approaches to business-university interaction. It places an emphasis on raising awareness outside the mathematical community of the benefits of adopting a mathematical way of thinking, often transferring ideas across application domains to create fresh insights that deliver real value.

www.smithinst.co.uk

The LMS-Smith Knowledge Transfer Reports are an initiative that is coordinated jointly by the Smith Institute and the Computer Science Committee of the LMS. The reports are being produced as an occasional series, each one addressing an area where mathematics and computing have come together to provide significant new capability that is on the cusp of mainstream industrial uptake. They are written by senior researchers in each chosen area, for a mixed audience in business and government. The reports are designed to raise awareness among managers and decision-makers of new tools and techniques, in a format that allows them to assess rapidly the potential for exploitation in their own fields, alongside information about potential collaborators and suppliers.

LONDON MATHEMATICAL SOCIETY

Smith institute
for industrial mathematics and system engineering