Formal Methods: Practice and Experience

JIM WOODCOCK
University of York

PETER GORM LARSEN
Engineering College of Aarhus

JUAN BICARREGUI
STFC Rutherford Appleton Laboratory
and
JOHN FITZGERALD
Newcastle University

Formal methods use mathematical models for analysis and verification at any part of the program life-cycle. We describe the state of the art in the industrial use of formal methods, concentrating on their increasing use at the earlier stages of specification and design. We do this by reporting on a new survey of industrial use, comparing the situation in 2009 with the most significant surveys carried out over the last 20 years. We describe some of the highlights of our survey by presenting a series of industrial projects, and we draw some observations from these surveys and records of experience. Based on this, we discuss the issues surrounding the industrial adoption of formal methods. Finally, we look to the future and describe the development of a Verified Software Repository, part of the worldwide Verified Software Initiative. We introduce the initial projects being used to populate the repository, and describe the challenges they address.

Categories and Subject Descriptors: D.2.4 [**Software/Program Verification**]: Assertion checkers, Class invariants, Correctness proofs, Formal methods, Model checking, Programming by contract; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Assertions, Invariants, Logics of programs, Mechanical verification, Pre- and post-conditions, Specification techniques; F.4.1 [**Mathematical Logic**]: Mechanical theorem proving; I.2.2 [**Automatic Programming**]: Program verification.

Additional Key Words and Phrases: Experimental software engineering, formal methods surveys, Grand Challenges, Verified Software Initiative, Verified Software Repository.

1. INTRODUCTION

Formal methods are mathematical techniques, often supported by tools, for developing software and hardware systems. Mathematical rigour enables users to analyse and verify these models at any part of the program life-cycle: requirements engineering, specification, architecture, design, implementation, testing, maintenance, and evolution.

The vital first step in a high-quality software development process is requirements

Corresponding Author's address: Jim Woodcock, Department of Computer Science, University of York, Heslington, York YO10 5DD, UK; email: jim@cs.york.ac.uk.

engineering. Formal methods can be useful in eliciting, articulating, and representing requirements [George and Vaughn 2003]. Their tools can provide automated support needed for checking completeness, traceability, verifiability, and reusability, and for supporting requirements evolution, diverse viewpoints, and inconsistency management [Ghose 2000].

Formal methods are used in specifying software: developing a precise statement of what the software is to do, while avoiding constraints on how it is to be achieved. Examples of these methods include ASM [Börger and Stärk 2003], B [Abrial 1996], and VDM [Jones 1990]. A specification is a technical contract between programmer and client to provide them both with a common understanding of the purpose of the software. The client uses the specification to guide application of the software; the programmer uses it to guide its construction. A complex specification may be decomposed into sub-specifications, each describing a sub-component of the system, which may then be delegated to other programmers, so that a programmer at one level becomes a client at another (*design by contract* [Meyer 1991]).

Complex software systems require careful organisation of the architectural structure of their components: a model of the system that suppresses implementation detail, allowing the architect to concentrate on the analyses and decisions that are most crucial to structuring the system to satisfy its requirements [Allen and Garlan 1992; van Lamsweerde 2003]. WRIGHT is an example of an architectural description language based on the formalisation of the abstract behaviour of architectural components and connectors [Allen 1997].

Formal methods are used in software design. Data refinement involves state machine specification, abstraction functions, and simulation proofs; see the early paper by Hoare [Hoare 1972], its central role in methods like VDM [Jones 1990], and in program refinement calculi [Dijkstra 1975; Morris 1987; Morgan 1988; Back and von Wright 1990].

At the implementation level, formal methods are used for code verification. Every program-specification pair implicitly asserts a correctness theorem that, if certain conditions are satisfied, the program will achieve the effect described by its documentation. Code verification is the attempt to prove this theorem, or at least to find out why the theorem fails to hold. The inductive assertion method of program verification was invented by Floyd and Hoare [Floyd 1967; Hoare 1969], and involves annotating the program with mathematical assertions, which are relations that hold between the program variables and the initial input values, each time control reaches a particular point in the program. Code can also be generated automatically from formal models; examples include the B-method [Abrial 1996] and SCADE [Berry 2008], both discussed in Sects 4.2 and 4.5.

It is natural that formal methods should underlie principled testing methods, and Gaudel has established this as an important research topic [Gaudel 1995]. Hoare describes the use of formal assertions in Microsoft, not for program proving, but for testing [Hoare 2002a]. A survey of current research in formal aspects of testing is in [Hierons et al. 2008]. Formal methods are used in software maintenance [Younger et al. 1996] and evolution [Ward and Bennett 1995]. Perhaps the widest application of formal methods is in the maintenance of legacy code: in some of Microsoft's most successful products, every tenth line is an assertion [Hoare 2002b].

In this paper, we assess the current state of the art in the industrial application of formal methods, concentrating on their increasing use at the earlier stages of specification and design. We first revisit several influential surveys of the use of formal methods and verification technology in industry (Sect. 2). We then present the results of a new survey of industrial practice in Sect. 3; this is the most comprehensive survey ever published, and gives us a view of how industrial application has changed over the last 20 years. In Sect. 4, we describe selected industrial projects from the last 20 years, representing a cross-section of applications including national infrastructure, computer microcode, electronic finance, and security applications. Sect. 5 contains our observations about the current state of the art, based on the survey findings and highlighted projects. A weakness in the current situation is lack of a substantial body of technical and cost-benefit evidence from applications of formal methods and verification technology; in Sect. 6, we describe the Verified Software Repository that is being built in response to this challenge. Finally, in Sect. 7, we draw some conclusions about current practice and experience. A list of acronyms is provided as an appendix.

2. SURVEYS OF FORMAL METHODS PRACTICE

The transfer of formal methods technology into industry has been an objective for researchers and practitioners for several decades. The potential benefits for reduced defect densities in specifications, designs, and code have to be achieved at reasonable cost and within the constraints of real industrial settings. By the early 1990s, questions were being asked about whether formal methods could ever be viable parts of industrial development processes. Several significant surveys from that time identified challenges to verification practice and experience that subsequent research has sought to address. We briefly review some of the major publications surveying the state of industrial application.

Hall's defence of formal methods as an engineering approach identifies seven "myths" about formal methods [Hall 1990]. Wing explained the underlying concepts and principles for formal methods to newcomers [Wing 1990]. Thomas presented evidence for the cost effectiveness of industrial use of formal methods from a CEO's perspective [Thomas 1992]. Austin carried out a survey into the industrial use of formal methods, in order to discover the reasons for their rather low acceptance in industry [Austin and Parkin 1993]. Austin used a questionnaire to assess the uses made of formal methods in both research and application, and to gather opinions on the barriers to wider industry adoption. A majority of responses analysed (126) reported on the use of model-oriented formalisms (such as VDM) and concentrated on specification rather than verification.

Craigen and his colleagues surveyed the application of formal methods, aiming at providing an authoritative record of the applications of formal methods [Craigen et al. 1993a; 1993b]. Their survey covered 12 case studies, each based on an application of formal techniques in an industrial setting. A combination of questionnaires, literature reviews, and interviews was used to derive information on each application. The range of formalisms included model-oriented approaches, a process calculus (CSP), and verification environments. None of the studies addressed systems bigger than around 300 KLOC, and the majority were much smaller (mostly

high-integrity applications). The survey came to positive conclusions about the improving maturity of formal methods and the fact that they had been applied to significant systems. Regulatory support for enhanced software engineering practices was important in providing increased motivation for adoption. They observed that "Tool support, while necessary for the full industrialisation process, has been found neither necessary nor sufficient for successful application of formal methods to date." Nevertheless, Bloomfield noted that the immaturity of theories and tool bases meant that some successful applications require a "heroic" level of effort, that tools are developed but not transferred across platforms or research groups, and that tools are not always accompanied by advances in methodology or theory [Bloomfield and Craigen 1999].

Rushby produced a technical report for NASA, explaining to stakeholders from the aerospace domain what formal methods are and how they can be applied in the development and certification of critical systems [Rushby 1993]. The perception of a long-term crisis in software development motivated a wider interest in the potential of verification [Gibbs 1994; Cuadrado 1994]. Bowen and Hinchey's article [Bowen and Hinchey 1995] is similar to [Hall 1990], but in a humorous fashion, stating commandments that shall be followed when applying formal methods; this was revisited in [Bowen and Hinchey 2006]. They also edited a book containing a collection of 15 different applications of formal methods using different formalisms [Hinchey and Bowen 1995]. Hinchey and Bowen [Hinchey and Bowen 1996] felt that standards, tools, and education would "make or break" industrial adoption, while Glass [Glass 1996] saw a chasm between academics who "see formal methods as inevitable" and practitioners who "see formal methods as irrelevant". Other articles cite weaknesses in notations, tools and education as challenges to wider acceptance of formal methods technology.

In spite of the optimistic conclusions of some surveys, a round-table article in *IEEE Software* in 1996 [Saiedian 1996] showed the divergence of opinion on whether formal methods were delivering hoped-for improvements in practice. Clarke and Wing's article was the output of a working group, and it gave a brief introduction to the notions in formal methods, listed notable industrial applications, and recommended future directions for the formal methods community [Clarke and Wing 1996]. Kelly's technical reports were orchestrated by NASA and formed a guidebook on the use of formal methods for specification and verification of software and computer systems [NASA 1998] and [NASA 1997].

Bloomfield's wide-ranging review [Bloomfield and Craigen 1999] includes evaluations of research programmes, major conferences, and industrial application areas. A leading point is the suggestion that models of technology diffusion should consciously be applied to formal methods adoption. Although they saw significant take-up in critical application domains, the authors identified several reasons for the general failure to adopt formal techniques.

The surveys take very different viewpoints. Some, such as Craigen's, base conclusions on analyses of a selected group of applications. Others, such as Austin's, have a wider ranging view of industry and academia. Still others, such as Bloomfield's, use reviews of whole research programmes. In spite of the differences in approach, there is some agreement on significant challenges to successful industrial adoption.

In the next section, we present a new quantitative survey of industrial practice in formal methods. In Sect. 5, we compare the results of the new survey with the major finding of the previous surveys.

3. A SURVEY OF CURRENT USE AND TRENDS

In order to help gain further understanding of trends and advances against the challenges identified in the papers described above, we undertook a survey to gather information in a consistent format from a number of industrial projects known to have employed formal techniques. We sent an open invitation to as many communities as we could to participate in our survey. This may have biased it towards those with whom we have the strongest contacts; we have, for instance, few contributions on model checking. In spite of this, the uniform way in which the data was collected does allow comparisons between projects, and gives some insight into current practice and long-term trends in the use of formal methods.

Using a structured questionnaire, data was collected between November 2007 and December 2008 on 62 industrial projects known from the published literature, mailing lists, and personal experience to have employed formal techniques. The projects surveyed came (in decreasing order) from Europe, Northern America, South America, Australia, and Asia. If an individual had experience of more than one project, then separate questionnaires were completed for each project. In 56 of the projects, data was collected directly from individuals who had been involved those projects, and in the remaining six cases we used information available in the literature.

3.1 Data Collected

Figure 1 presents the application areas to which the projects related. The largest

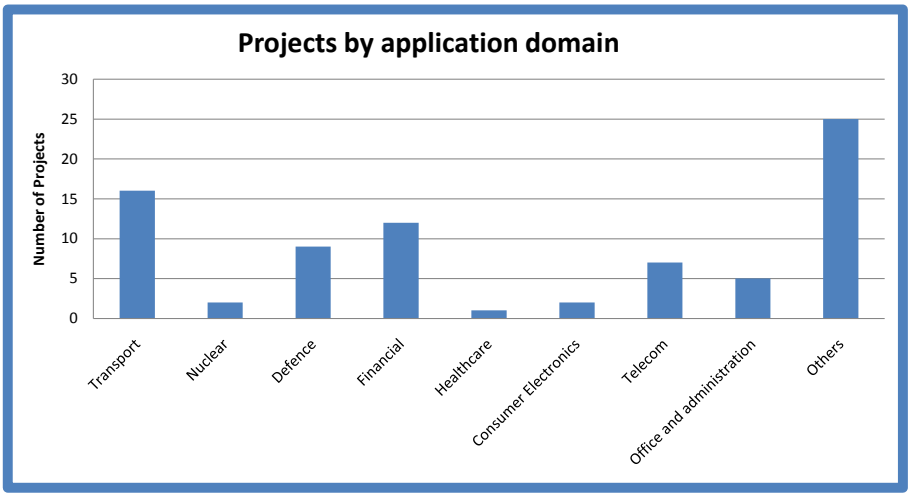

Fig. 1. Application Domains

single application domain was transport, followed by the financial sector. Other major sectors were defence, telecommunications, and office and administration. Other

areas with only one or two responses were: nuclear, health care, consumer electronics, space, semantic web, resource planning, automated car parking, embedded software, engineering, and manufacturing. Some 20% of responses additionally indicated that the projects related to software development tools themselves, such as operating systems, compilers, and CASE tools, and a further 10% related to computing applications within the domain, such as high-performance computing, runtime code optimisation, file system replication, access control, communications protocols, and microcomputer design.

The types of applications are presented in Figure 2. The largest groups were

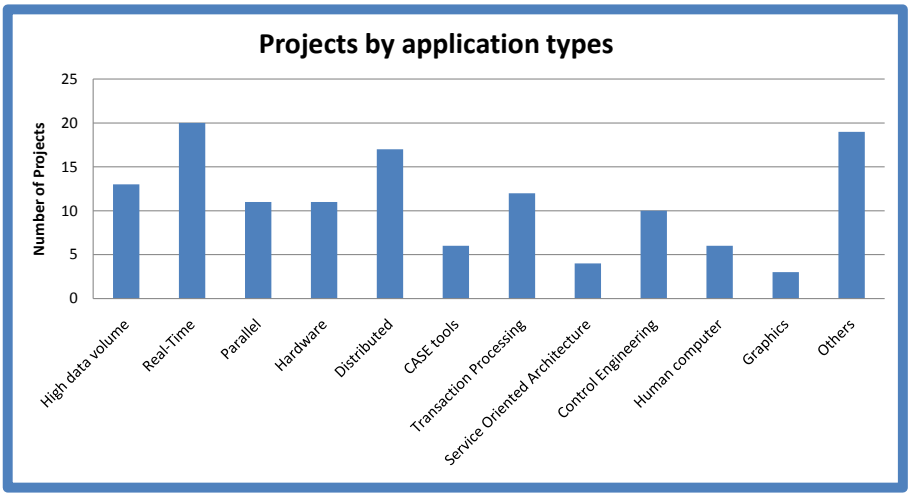

Fig. 2. Application Types

real-time applications, distributed applications, transaction processing and high data volume. Others included parallel programming, hardware, control engineering, HCI, service-oriented computing, and graphics. Certification standards were indicated as applying in 30% of responses, notably the International Electrotechnical Commission's IEC 61508, and the Common Criteria and UK Level E6 for Information Technology Security Evaluation. Others included CENELEC EN50128 for railways, DO-178B Level A for avionics, UK MoD software Defence Standards 00-55 and 00-56, and IEEE Standard 754 for Floating Point Numbers.

Figure 3 presents the start dates of the projects. After taking account of the fact that ongoing projects are unlikely to be represented in the survey, it would seem that the frequency of projects is higher in recent years, although this could simply be a reflection of the way that the data was collected, with more recent projects being more responsive.

As shown in Figure 4, 50% of respondents gave an indication of the size of the software in terms of lines of code. Of these, the split was roughly equal on a logarithmic scale between 1–10 KLOC, 10–100 KLOC, and 100–1000 KLOC.

Figure 5 presents the rates of use of various techniques, such as specification

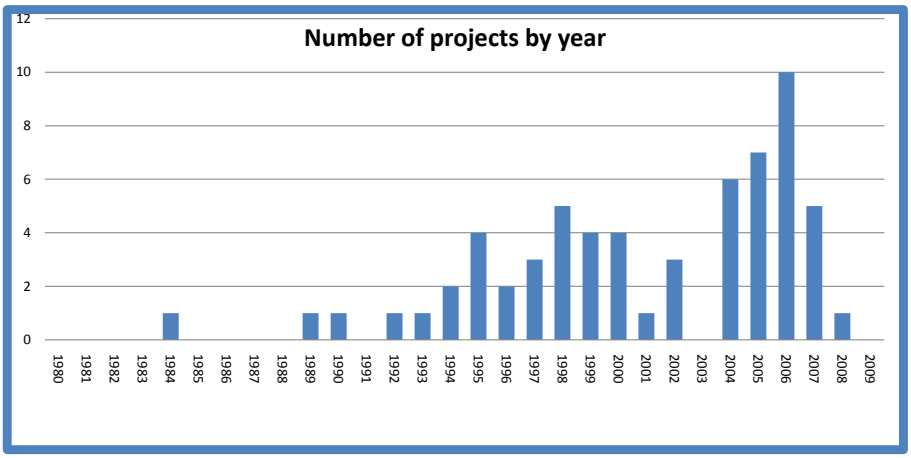

Fig. 3. Project start date

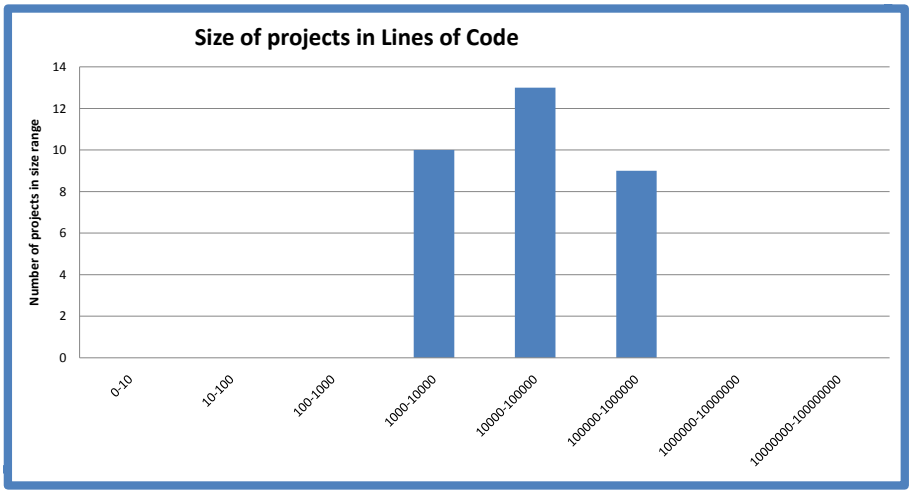

Fig. 4. Project size in Lines of Code

and modelling, execution (specification interpretation and testing), inspection (of specification and model).

We looked for dependencies between application domain and type of software developed, but found only one significant correlation, which was a high representation of transaction processing software in the financial domain (Fisher's exact test, double tailed, $p < 0.01$).[1] Without comparison to general data relating software type and application domain, this may be assumed to reflect a trend in software as

[1]Fisher's exact test is a statistical significance test used in the analysis of the relationship between two or more variables, where sample sizes are small.

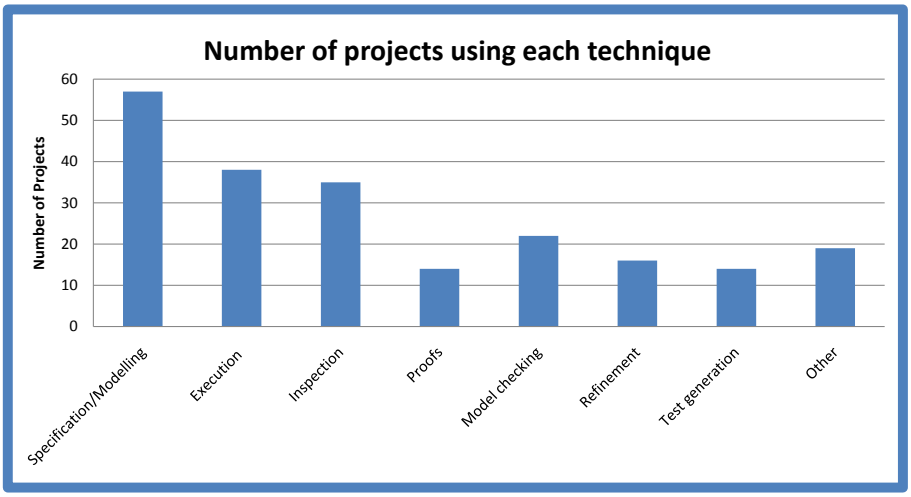

Fig. 5. Techniques used

a whole, rather than being related to the application of formal techniques. Similarly, no significant dependencies at all were found between the techniques used for different application domains, and only a few, reasonably mild, correlations were observed between the techniques used for different types of software. These latter correlations demonstrate a higher than average use of model checking in consumer electronics and of inspection in transaction processing software (Fisher's Exact Test, double tailed, $p = 0.03$ for each).

On the other hand, on correlating the techniques used against the project date, we found that the use of model checking has increased greatly from 13% in the 1990s to 51% in this decade. This is a highly significant change (Fisher's exact test, double tailed, $p = 0.003$). In contrast, no significant change was found in this period for the use of proof, refinement, execution or test case generation.

When asked to indicate which roles were part of the project team from a predefined list (product management, program management, development, user experience, tester, release management, architect, other), the largest responses were "tester" (50%) and "architect" (46%) with all other responses being under 10%. Regarding previous expertise in the techniques used, 40% reported "considerable previous experience", 45% reported "some previous experience", and 22% reported "no previous expertise". The total adds up to more than 100% because some respondents reported mixed teams of more than one category. Of those reporting "no previous expertise", one half were in a mixed team with more experienced colleagues; the others were introducing techniques to a team not previously experienced in these techniques. Regarding training, 33% reported "no training given", 54% reported "some training given", and 5% reported "considerable training given".

3.2 Outcomes—the Effect on Time, Cost, and Quality

Figure 6 shows the overall effect of the use of formal techniques on time, cost, and quality. The effect on time taken to do the work was on average beneficial.

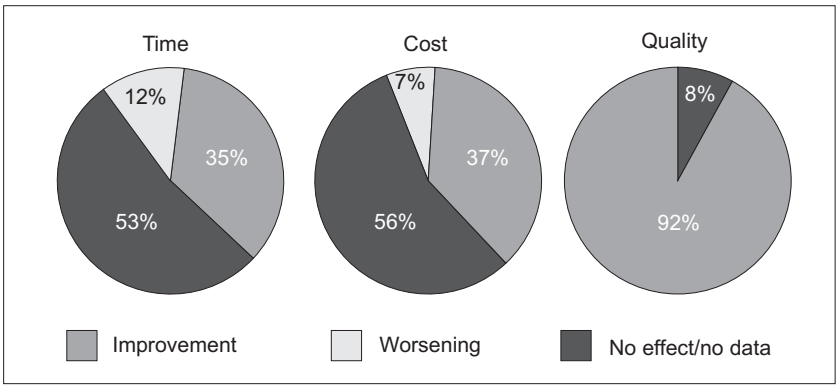

Fig. 6. Did the use of formal techniques have an effect on time, cost, and quality?

Three times as many reported a reduction in time, rather than an increase. Many responses indicated that it was difficult to judge the effect on time taken, although several noted increased time in the specification phase, which may or may not have been compensated for by decreasing time later. For example:

> "Difficult to judge but extensive specification phase probably added elapsed time, although it may have saved time later."
>
> "Long specification phase probably added to elapsed time."
>
> "Modest increase in time spent during early design... was recovered many times over during system integration and testing."

Of cases expressing a view, five times as many projects reported reduced costs as those that reported increased costs. Some notable comments with respect to the effect on cost include:

> "We observed a substantial improvement in productivity once the code generation subsystem of the tool had been bootstrapped, due to the use of code generation from specifications..."
>
> "The cost increase was largely due to the lack of precise, complete information about the required externally visible behavior of the software product... Once the code was implemented the required behavior was clear, and applying formal specification and formal verification was relatively straightforward. The one expensive part of the code verification process was the annotation of the code with pre- and postconditions. Once the annotated code was available, showing the correspondence between the annotated code and the abstract specification of the required behavior was straightforward. This latter process included adding more annotations and correcting annotations that were incorrect. During this process, the abstract specification and the required security properties changed very little."

In contrast, the use of formal techniques is believed by respondents to have improved quality, with 92% of all cases reporting an increase in quality compared to other

techniques, and no cases reporting a decrease in quality. Most were related to the detection of faults (36%). Other common reasons given for improvement were: improvements in design (12%), increased confidence in correctness (10%), improved understanding (10%), and early identification of faults or other issues (4%).

3.3 Respondents' Conclusions

As shown in Figure 7, respondents have generally been positive about the successful use of formal methods, although, due to the bias in selection described above, one would expect stakeholders that did not see value in the use of formal methods to be under-represented among the responses. Figure 8 illustrates that the respondents in general were satisfied with the formal techniques used in their projects, whereas Figure 9 shows that in 9% of the projects, the tools applied have not fully been able to live up to expectations. Finally, Figure 10 demonstrates that a majority of the respondents wish to use a similar technology again on new projects.

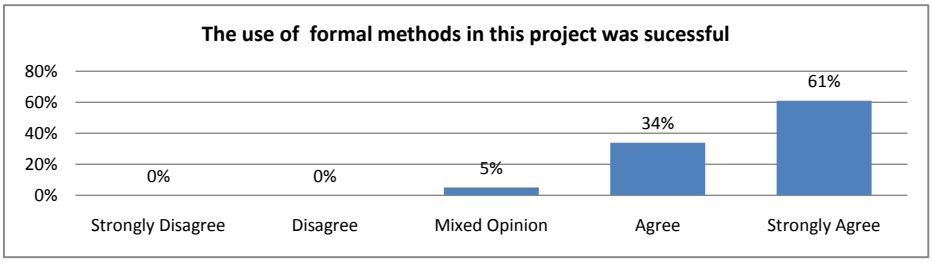

Fig. 7. Overall satisfaction with the used of formal techniques

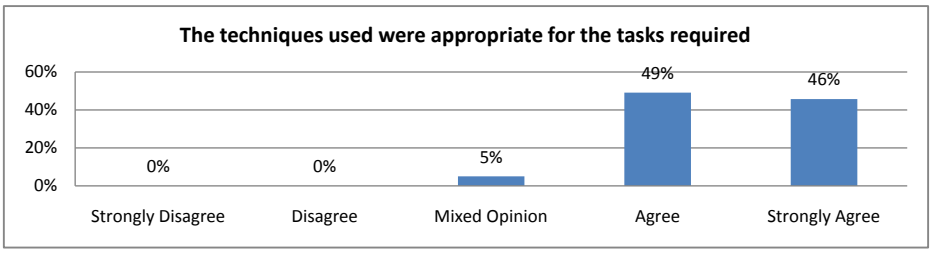

Fig. 8. Overall satisfaction with the techniques used

3.4 Discussion on the Results of the Survey

Take-up by users. The survey shows that the take up of formal techniques is distributed across a wide range of application domains, including a considerable number related to the production of the software development tools themselves. This reflects the fact that many of the tools used are general-purpose, as programming languages are, so research on generic methods (that is independent of application area) is still relevant and should proceed alongside development of more special-purpose generators.

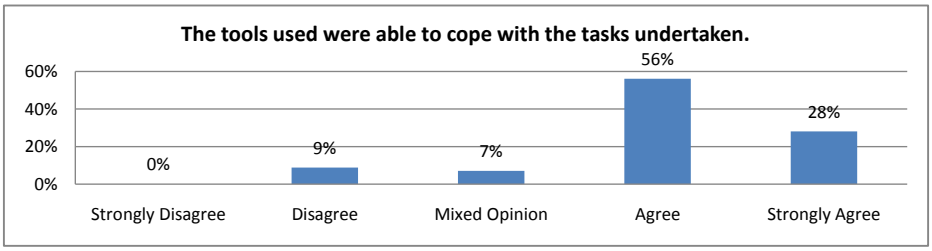

Fig. 9. Overall satisfaction with the tools used

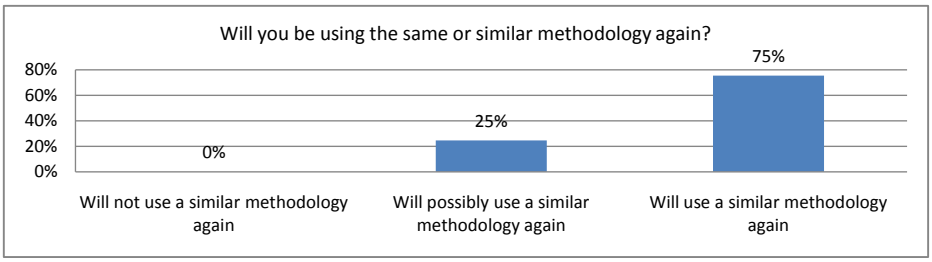

Fig. 10. Intention to use formal techniques again

Tools and techniques. Some projects undertook extensive and comprehensive analysis, significant manual intervention in the formal analysis, and therefore a major change to existing informal techniques. On the other hand, some projects managers were more interested in broad rather than deep analysis, hoping for some benefit from little or no manual effort.

It is clear that Moore's law has had a very significant impact on the availability of computational power over the two decades spanned by the projects reported in the survey, with typical desktop computing and storage resources increasing by perhaps some 10,000 times in that period. Together with significant theoretical advances, this has had a major impact on the scale of problem that can be addressed by automated techniques, such as model checking, and would seem to indicate a continuing increase in the level of automation and move towards formal methods disappearing "under the hood" in due course (see [Tiwari et al. 2003] for a discussion of this movement). Such a trend is epitomised by the following quote from a project that employed significant levels of model checking:

> "To be useful during product development, formal methods needs to provide answers in seconds or minutes rather than days. Model-checking can do this very effectively when applied to the right kinds of system designs. To take advantage of this, model-checking has to be tightly integrated into the commercial design and verification tools that developers are already using."

To what degree, and in what timescale, will mathematical proof succumb to Moore's law? One respondent reflected on the tools available as follows:

> "Tools for formal methods are still very weak compared with what is theoretically possible."

The effect on time, cost, and quality. Of those expressing an opinion, significantly more projects are seen to have reduced timescales and costs and improved quality by using formal techniques. But it is perhaps surprising that in a majority of projects surveyed, there is no data available on relative time and costs. On the other hand, the overall satisfaction with their use among the respondents is clear (Figures 7 and 10).

> "No precise measurements have been done. By rule-of-thumb it was estimated that the design effort exceeded a conventional effort by about 10%. Presumably maintenance costs were reduced drastically. But this judgement is based only on hypothetically assumed fault rates of software developed conventionally."

4. HIGHLIGHTED PROJECTS

We present a series of industrial formal methods projects, chosen to show a cross-section of work over the last two decades. The emphasis is on developing verified systems cost-effectively, and the applications include microcode and firmware, railways, national infrastructure, smart cards, and a biometric-based security application. From Sect. 4.1 onwards, each project is included in the survey described in the last section, but we start with a review of the range of verification tools reported in the literature as having been used in industry.

There is a trade-off between the level of automation achievable in verification and the complexity of the analysis performed. At one end of the range there are static analysers that identify potential program defects, such as variable use before definition, constant conditions, and possibly out-of-range expressions. Examples include heuristic tools such as the 30-year-old UNIX utility `lint` [Johnson 1978], which flags potential defects in C programs, and more recent tools such as Splint [Evans and Larochelle 2002], which provides annotation-assisted lightweight static checking. These kinds of program analysis tools are in use in industry; see [Hoare 2002a] for a description of the instrumentation of legacy Microsoft code with assertions for use in massive regression testing, and [Larus et al. 2004] for a short survey of the use of tools within Microsoft. Modern analysers such as PREfix [Bush et al. 2000] and PREfast can analyse multi-million-line C and C++ programs for potential null-pointer dereferences, improper memory allocation and deallocation, uninitialised variable use, simple resource-state errors, improper library use, and problematic programming idioms. Such tools often use formally unsound analytical techniques and so may produce false positives (identifying spurious errors), although efforts are made to minimise these. The SLAM tool on the other hand can certify a program free from a particular kind of error [Ball and Rajamani 2002] by checking that API usage follows certain sequencing rules given by the user. It has been used to find many bugs in Windows device drivers, and is distributed as the Static Driver Verifier, part of the Windows development kit. The ESP tool [Das et al. 2002] is similar to SLAM, but trades precision against large-scale tractability for analysing huge code bases; it has been used to find buffer over-runs in large-scale system code and to validate an OS kernel's security properties.

Further along the verification spectrum there are tools such as model checkers and abstract interpreters. The SPIN automata-based model checker has been used

to find logical design errors in distributed systems, such as those used on the Deep Space 1, Cassini, Mars Exploration Rovers, and Deep Impact space missions [Holzmann 2004]. SPIN checks the logical consistency of a specification, reporting on deadlocks, unspecified message receptions, incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. An example of an abstract interpreter is the ASTRÉE real-time embedded-software static analyser [Blanchet et al. 2003], which tries to prove the absence of all run-time errors in a C program. It does this completely automatically for the primary flight-control software for the Airbus A340 and the electric flight-control codes for the A380.

At the furthest end of the verification spectrum, theorem provers can verify arbitrary conjectures in a given logic with varying degrees of automation. For example, Vampire is an automatic theorem prover for first-order classical logic [Riazanov and Voronkov 2002]. KIV, on the other hand, is an interactive theorem prover with a user-definable object logic [Balser et al. 2000], and it has been used in a wide range of applications, from verifying protocols for medical procedures [Hommersom et al. 2007] to verifying protocols for smart cards [Haneberg et al. 2008].

4.1 The Transputer Project

The Transputer series of microprocessor chips were designed specifically for parallel processing [Inmos Ltd 1988b]. Gibbons describes the development of one of the Transputers: the T800 floating-point unit [Gibbons 1993], which combined a 32-bit reduced instruction set CPU, some memory, four bidirectional communications links, and a floating-point arithmetic unit on a single chip. Its successor, the T9000, was rather more sophisticated, with richer connectivity, memory model, and pipelined processor. A Transputer based on T9000 technology, the ST20, is still very widely used in chip-sets for set-top box and GPS applications. The programming language for the Transputer is Occam [Inmos Ltd 1988a; Jones and Goldsmith 1988], a simple, low-level, executable subset of CSP [Hoare 1985].

Inmos started to develop the T800 in 1986, using a conventional approach that required months of testing, since floating-point units are notoriously complex devices and prone to design bugs. As the extent of the required testing became clear, work started on the formal development of a correct-by-construction floating-point unit [Shepherd 1988; Barrett 1987; 1989; Shepherd and Wilson 1989; Barrett 1990; May et al. 1992]. The natural language IEEE-754 standard for floating-point arithmetic [IEEE 1985] was formalised in Z [Spivey 1989; Woodcock and Davies 1996]. The specification is described in [Barrett 1987; 1989], and revealed some problems in the standard. For example, the standard requires that diagnostic information about the results of invalid operations (such as the square root of a negative number) be propagated through further operations. But this is not always possible. The next task was to show that a floating-point package written in Occam and used in previous Transputers was a correct implementation of IEEE-754. The attempted verification using Hoare logic [Hoare 1969] revealed errors in rounding and remainder operations. Barrett later remarked to Gibbons that "it was only a very small class of test vectors that would have shown up the errors" [Gibbons 1993]. With corrections in place, the Occam package was verified correct and used as an intermediate representation of the functionality of the required microcode. It was too abstract to be directly useful for hardware design, so the Occam transformation

system [Goldsmith et al. 1987] was used to apply the laws of Occam programming [Roscoe and Hoare 1988] to produce an equivalent microcode program.

The development of the floating-point unit, from natural language to silicon, was at least three months faster than the alternative informal development that ran concurrently [Barrett 1990]. Each month's delay in production was estimated to cost US$1M [Barrett 1990]. Gibbons reports that exactly two bugs have been found in the floating-point microcode [Gibbons 1993]. The first was introduced by the translation program that converted the micro-level Occam into assembly code for the chip; the second was introduced by a hand optimisation of this assembly code. Oxford University and Inmos jointly received a Queen's Award for Technological Achievement in 1990 "to recognise and encourage outstanding achievements advancing process or product technology in the UK".

4.2 Railway Signalling and Train Control

In 1988, GEC Alsthom, MATRA Transport, and RATP (the Parisian public transport operator) started working on a computerised signalling system for controlling the RER regional express network commuter trains in Paris (reported in [Bowen and Stavridou 1993]). The objective of the project was to increase network traffic by 25%, while preserving existing safety levels. The resulting SACEM system with embedded hardware and software was delivered in 1989 and has controlled the speed of all trains on RER Line A in Paris, involving seven billion passenger journeys, since its introduction.

The SACEM software consists of 21,000 lines of Modula-2 code, of which 63% is regarded as safety-critical and has been subjected to formal specification and verification [Guiho and Hennebert 1990; Hennebert and Guiho 1993]. The specification was constructed in B [Abrial 1996] and the proofs were done interactively using automatically generated verification conditions for the code. The validation effort for the entire system (including non-safety-critical procedures) was about 100 man-years. Hennebert and Guiho [Guiho and Hennebert 1990] claim that the system is safer as a result of the formal specification and verification exercise. The project team reported a difficulty in communication between the verifiers and the signalling engineers, who were not familiar with the B-method. This was overcome by providing the engineers with a French description derived manually from the formal specification. The SACEM system is further described in [Guiho and Hennebert 1990], which presents various dependability requirements and their implementation. Techniques to ensure safety include on-line error detection, software validation, and fault tolerance of the onboard-ground compound system.

Abrial (the creator of B) reports on two further railway projects carried out using B [Abrial 2007]: Line 14 of the Paris Métro, a system in use since October 1998 [Behm et al. 1999]; and the driverless Paris Roissy Airport shuttle, in use since 2007 [Badeau and Amelot 2005]. Only the safety-critical parts were developed using B, representing one-third of each software system. Table I (taken from [Abrial 2007]) shows the main characteristics of the two systems. Since Line 14 is completely automatic, the safety-critical part concerns the running and stopping of trains and the opening and closing of train and platform doors. No unit tests were performed for the Line 14 or Roissy Shuttle projects; they were replaced by some global tests that were all successful. This reduced the overall costs significantly.

	Paris Métro Line 14	**Roissy Shuttle**
Line length (km)	8.5	3.3
Number of stops	8	5
Inter-train time (s)	115	105
Speed (km/hr)	40	26
Number of trains	17	14
Passengers/day	350,000	40,000
Number of lines of Ada	86,000	158,000
Number of lines of B	115,000	183,000
Number of proofs	27,800	43,610
Interactive proof percentage	8.1	3.3
Interactive proof effort (person-months)	7.1	4.6

Table I. Statistics for the Paris Métro Line 14 and Roissy Shuttle projects.

4.3 Mondex Smart Card

In the early 1990s, the National Westminster Bank and Platform Seven[2] developed a smartcard-based electronic cash system, Mondex, suitable for low-value cash-like transactions, with no third-party involvement, and no cost per transaction. A discussion of the security requirements can be found in [Stepney et al. 2000; Woodcock et al. 2008]; a description of some wider requirements can be found in [Aydal et al. 2007]. It was crucial that the card was secure, otherwise money could be electronically counterfeited, so Platform Seven decided to certify Mondex to one of the very highest standards available at the time: ITSEC Level E6 [ITSEC 1991], which approximates to Common Criteria Level 7 [CCRA 2006]. This mandates stringent requirements on software design, development, testing, and documentation procedures. It also mandates the use of formal methods to specify the high-level abstract security policy model and the lower-level concrete architectural design. It requires a formal proof of correspondence between the two, in order to show that the concrete design obeys the abstract security properties. The evaluation was carried out by the Logica Commercial Licenced Evaluation Facility, with key parts of the work subcontracted to the University of York to ensure independence.

The target platform smartcard had an 8-bit microprocessor, a low clock speed, limited memory (256 bytes of dynamic RAM, and a few kilobytes of slower EEPROM), and no built-in operating system support for tasks such as memory management. Power could be withdrawn at any point during the processing of a transaction. Logica was contracted to deliver the specification and proof using Z [Spivey 1989; Woodcock and Davies 1996]. They had little difficulty in formalising the concrete architectural design from the existing semi-formal design documents, but the task of producing an abstract security policy model that both captured the desired security properties (in particular, that "no value is created" and that "all value is accounted for") and provably corresponded to the lower-level specification, was much harder. A very small change in the design would have made the abstraction much easier, but was thought to be too expensive to implement, as the parallel implementation work was already well beyond that point. The 200-page proof was

[2]Platform Seven is a software house spun out from NatWest; it is now owned by DataCard, a major shareholder in Gemplus, another smart card company.

carried out by hand, and revealed a small flaw in one of the minor protocols; this was presented to Platform Seven in the form of a security-compromising scenario. Since this constituted a real security problem, the design was changed to rectify it. The extensive proofs that were carried out were done manually, a decision taken at the time to keep costs under control. Recent work (reported below) has shown that this was overly cautious, and that Moore's Law has swung the balance further in favour of cost-effective mechanical verification.

In 1999, Mondex achieved its ITSEC Level E6 certificate: the very first product ever to do so. As a part of the ITSEC E6 process, the entire Mondex development was additionally subject to rigorous testing, which was itself evaluated. No errors were found in any part of the system subjected to the use of formal methods.

Mondex was revived in 2006 as a pilot project for the Grand Challenge in Verified Software (see Sect. 6). The main objective was to test how the state of the art in mechanical verification had moved on in ten years. Eight groups took up the challenge using the following formal methods (with references to a full discussion of the kinds of analysis that were performed in each case): Alloy [Ramananandro 2008], ASM [Haneberg et al. 2008], Event-B [Butler and Yadav 2008], OCL [Kuhlmann and Gogolla 2008], PerfectDeveloper, π-calculus [Jones and Pierce 2007], Raise [George and Haxthausen 2008], and Z [Freitas and Woodcock 2008]. The cost of mechanising the Z proofs of the original project was 10% of the original development cost, and so did not dominate costs as initially believed. Interestingly, almost all techniques used in the Mondex pilot achieved the same level of automation, producing similar numbers of verification conditions and requiring similar effort.

4.4 AAMP Microprocessors

Miller reports experiences from several projects at Rockwell Collins [Miller 1998]. One of their earliest experiments was to specify and refine a micro Real Time Executive μRTE in the RAISE notation, RSL [RAISE Language Group 1992; RAISE Method Group 1995]. This was not a successful project: the language was thought to be too complex and required substantial training, and the only tools available were syntax and type checkers, with no support for developing and managing proofs.

A subsequent experiment with SRI International, sponsored by NASA [Miller and Srivas 1995; Srivas and Miller 1995], formally verified the microcode for the AAMP5 microprocessor using PVS [Owre et al. 1992]. The AAMP5 is a proprietary microprocessor widely used in Rockwell Collins products. It has a stack-based architecture, a large instruction set, makes extensive use of microcode, has a pipelined architecture, and complex processing units. It contains approximately 500,000 transistors with performance between an Intel 386 and 486. Rockwell Collins specified the AAMP5 at both the register transfer and instruction set level, with a retrieve relation between the two to prove the correctness of microcode instructions. The main lesson learned from the AAMP5 project was that it was technically possible to prove the correctness of microcode, and that their engineers could read and write formal specifications.

Two errors were found in the AAMP5 microcode while creating the specification, and this convinced them that there is value in just writing a formal specification. But they also convinced themselves that mechanical proofs of correctness provide a very high level of assurance. They did this by seeding two very subtle errors in the

microcode that they delivered to SRI, and then waiting to see if they would find them. SRI did indeed discover the errors using a systematic process: the only way not to have found the errors would have been to fail to carry out the proofs.

The biggest problem for the AAMP5 project was that the cost was too high: more than 300 hours per instruction. This figure appears to have been inflated for a variety of reasons, including the steep learning curve using PVS for the first time and the need to develop many supporting application-oriented theories. They knew their costs would drop dramatically the next time around, but they could not predict by how much, so they undertook a second experiment [Miller et al. 1996], the verification of the microcode in the AAMP-FV, again sponsored by NASA and with SRI. The goal was to demonstrate dramatic reduction in cost through reuse of the AAMP5 infrastructure and expertise.

Significantly, the AAMP-FV project confirmed that the expertise gained on the AAMP5 project could be exploited to dramatically reduce the cost of formal verification. Of the 80 AAMP-FV instructions, 54 were proven correct, and the cost of their verification dropped by almost an order of magnitude from that of the AAMP5 project. But as more complex instructions were attempted, proof techniques first developed on the AAMP5 project broke down and new approaches had to be devised. This phase progressed more as an exploratory project, with a steep learning curve and unexpected delays. One of the main contributions of the AAMP-FV project was the development of methods to handle complex microcode.

The latest project involves the AAMP7 processor, which has a separation kernel built into the micro-architecture. Rockwell carried out a proof in ACL2 that a line-by-line model of the microcode adheres to a security policy about how partitions are allowed to communicate [Wilding et al. 2001; Greve and Wilding 2002]. What is hard about proofs of this kind is the complexity of dealing with pointer-laden data structures. The work received National Security Agency certification as a Multiple Independent Levels of Security device for use in cryptographic applications, at EAL-7 of the Common Criteria.

4.5 Airbus

Airbus have used SCADE for the last ten years for the development of DO-178B Level A controllers for the A340-500/600 series, including the Flight Control Secondary Computer and the Electric Load Management Unit. A summary of these and other industrial applications is described in [Berry 2008]. Esterel Technologies reports the following benefits: (i) A significant decrease in coding errors: for the Airbus A340 project, 70% of the code was generated automatically. (ii) Shorter requirements changes: the SCADE tool suite manages the evolution of a system model as requirements change, and in the Airbus A340 project, requirements changes were managed more quickly than before, with improved traceability. (iii) Major productivity improvement: Airbus reported major gains, in spite of the fact that each new Airbus project requires twice as much software as its predecessor.

Having achieved significant savings on the overall design cycle, Airbus adopted SCADE for A380 projects, where most on-board computers developed by Airbus and its suppliers benefit from SCADE technology. The SCADE Suite is used for the development of most of the A380 and A400M critical on-board software, and for the secondary flying command system of the A340-500/600 aircraft, in operational

use since August 2002. The A380 and A400M Cockpit Control and Display System and the On-board Airport Navigation System Display have been developed using SCADE Display, oriented towards the specification of graphical interfaces.

4.6 The Maeslant Kering Storm Surge Barrier

The *Maeslant Kering* is a movable barrier protecting the port of Rotterdam from flooding as a result of adverse weather and sea conditions. The decision to deploy and to reopen the barrier is made on the basis of meteorological data by a computer system. In terms of the international standard IEC 61508 [IEC 1997], the application was placed at Safety Integrity Level 4, for which the use of formal methods is "highly recommended". The developers (CMG) were deliberately cautious in defining goals for a formal methods deployment [Kars 1997; Tretmans et al. 2001; Wijbrans et al. 2008]. Refinement technology was not felt to be feasible for a system of this scale. It was also felt to be too high-risk an option to introduce several new techniques in one project. The approach was therefore to integrate modelling and verification technology within the normal design trajectory. The approach used formal modelling and verification in the analysis, design and realisation phases of system development. The focus of the formal work was the decision-making subsystem and its interfaces to the environment.

Data and operations were modelled in Z [Spivey 1989; Woodcock and Davies 1996], and this was embedded into a Promela model describing control, and designs were validated using the SPIN model checker [Holzmann 2004]. Promela and SPIN were selected because of the developers' prior experience with the tool and a perceived ease of use, meaning that the CMG engineers could perform most of the modelling and analysis work without having to bring in outside assistance. The Promela/Z linkage was *ad hoc* and did not itself have a formal semantics.

The final detailed design specified 29 programs with 20,000 lines of Z. Implementation was done via systematic coding in a safe subset of C++. There was no formal code verification. The final implementation was 200 KLOC for the operational system and 250 KLOC of supporting code (simulators, test systems, *etc.*). Informal but systematic test derivation from the Z model resulted in 80–90% code coverage for black box testing, with the remainder covered by white box tests. The problems raised during the process were logged; about 85% of them arose during development phases and around 15% during reliability and acceptance test. The residual faults have been minor. About half the faults detected during development were in code or design, the remainder being weaknesses in test specifications, configuration parameters or documentation.

The experience was largely positive, its report [Tretmans et al. 2001] deliberately echoing Hall's Seven Myths [Hall 1990]. The software was believed by the developers to be of significantly better quality than would otherwise have been achieved, and that this quality benefit would hold for non-critical systems also. A significant shift was noted in effort and cost towards specification and design phases. The authors noted that abstraction skills were an essential part of the modelling process and that the ease of constructing formal models should not seduce engineers away from devoting effort to selecting appropriate abstractions.

No major defects have been reported in the system developed using formal techniques. A mid-life upgrade was reported in 2008 [Wijbrans et al. 2008] and the

development of the successor application will continue to use formal techniques.

4.7 The Tokeneer Secure Entry System

The Tokeneer ID Station (TIS) project [Barnes et al. 2006], carried out by Praxis High Integrity Systems in conjunction with SPRE Inc., under the direction of NSA (National Security Agency), has shown that it is possible to produce high-quality, low-defect systems conforming to the Common Criteria requirements of Evaluation Assurance Level 5 (EAL5) [CCRA 2006]. The Tokeneer system was originally developed by the NSA to investigate various aspects of biometrics in access control, and consists of a secure enclave with controlled physical entry. Within the secure enclave are a number of workstations whose users have security tokens (*e.g.*, smartcards) in order to gain access to the workstations. Users present their security tokens to a reader outside the enclave, which uses information on the token to carry out biometric tests (*e.g.*, fingerprint reading) of the user. If the user passes these tests, then the door to the enclave is opened and the user is allowed entry. At entry time, the system adds authorisation information to the security token describing exactly the sort of access allowed for this visit to the enclave, such as times of working, security clearance, and roles that can be taken on.

Praxis completed MULTOS, an important project using formal methods, in 2002 (see [Hall and Chapman 2002]. MULTOS is a Multi-Application Operating System that allows several applications to reside on a single smart card. The success of this project led to a proposal by the NSA for a demonstrator project in secure software engineering. Praxis undertook this development over nine months in 2003, and a conference paper was eventually cleared for publication [Barnes et al. 2006]. The Tokeneer project material [Tokeneer 2009] was released under an NSA Technology Transfer Agreement in July 2008 as a contribution to the Verified Software Grand Challenge (see Sect. 6).

The TIS project re-developed one component of the Tokeneer system. To facilitate the development, TIS device simulators implemented by the independent reliability consultants (SPRE Inc.) were used in place of actual TIS devices. The core functionality of the system was written in SPARK, a subset of Ada with an accompanying tool-set, which was specifically designed for writing software for high-integrity applications [Barnes 2003]. The support software to interface it to simulated peripherals was written in full Ada. Tables II and III, taken from [Barnes et al. 2006], record the project's size, productivity, and effort. The project re-

	Size/source lines		**Productivity (LOC/day)**	
	Ada	SPARK annotations and comments	During coding	overall
TIS core	9,939	16,564	203	38
Support software	3,697	2,240	182	88

Table II. Tokeneer: size and productivity.

quired 260 person-days of effort, comprising three part-time staff working over nine months. The number of defects found in the system during independent system

Project phase	Effort %	Effort Person-days
Project management	11	28.6
Requirements	10	26.0
System specification	12	31.2
Design Core functions	15	39.0
TIS Core code and proof	29	75.4
System test	4	10.4
Support software and integration	16	41.6
Acceptance	3	7.8
Total	**100**	**260.0**

Table III. Tokeneer: breakdown by project phase.

reliability testing and since delivery in 2003 is two. One was discovered by code inspection after the completion of the project (see Spinellis's blog for an account of finding this bug [Spinellis 2008]).

A second bug was found by Praxis when they examined an undischarged proof obligation. It relates to code that validates integer values read from a file. These integers represent seconds, and are converted into tenths-of-seconds, which can cause an overflow error. The SPARK tools were used to generate verification conditions for partial correctness and run-time errors, but without side-conditions relating to Adas `Overflow_Check`, because of limited capability to discharge such VCs. Following improvements to the tools, the VCs were re-generated and checked, discovering the bug. The developers note [Chapman 2008] that the defect was not discovered by any testing during the original project, or any use or attempt to analyse the system since the initial delivery.

Barnes reports [Barnes et al. 2006] that the testing team from SPRE Inc. actually discovered two in-scope failures as part of their testing regime: both concerned missing items from the user manual, rather than errors in the TIS Core. The entry in Table III for system test does not include the testing contribution from SPRE Inc. Barnes estimates [Barnes et al. 2006] that a more representative figure might be 25%. The functional specification, written in Z and explanatory English, consists of about 100 pages.

The task set by NSA was to develop a system in conformance with the requirements in the Common Criteria for EAL5. In fact, Praxis exceeded the EAL5 requirements in a number of areas, because they found that it was actually cost-effective to use some of the more rigorous techniques. Praxis met the EAL5 criteria in the main body of the core development work, covering configuration control, fault management, and testing. They exceeded EAL5, coming up to EAL6 or EAL7 levels, in the development areas covering the specification, design, implementation, and correspondence between representations. Other aspects were out of scope, such as delivery and operational support.

4.8 The "Mobile FeliCa" IC Chip Firmware

"FeliCa" is a contactless IC card technology widely used in Japan, developed and promoted by Sony Corporation. Mobile telephones with FeliCa chips can serve as electronic purses, travel tickets, door keys, *etc.* FeliCa Networks Inc. decided to

use VDM++ and VDMTools [Fitzgerald et al. 2008] for the development of the firmware for a new generation IC chip containing new features but nevertheless operating to the strict timing requirements provided by earlier generations.

The project lasted three years and three months, and involved 50 to 60 people with an average age of a little over 30 years. No members had knowledge of or experience with the formal method at the time of project launch. VDM++ training (in Japanese) was provided for the development team by CSK. In addition an external VDM consultant from CSK Systems was used throughout the project. The new version of the firmware was subsequently released in millions of IC chips [Kurita et al. 2008; Larsen and Fitzgerald 2007].

A large number of VDM++ test cases were developed and then executed using the VDMTools interpreter. Using the VDMTools test coverage analysis facility, it was possible to display test coverage information on the VDM++ model after executing the entire test suite. Here, 82% of the VDM++ model was covered, and the remaining parts of the model were manually inspected. The main outputs included a 383-page protocol manual written in Japanese, a 677-page external specification document written in VDM++ (approximately 100 KLOC including comments, of which approximately 60 KLOC are test cases formulated in VDM++). The implementation was approximately 110 KLOC of C/C++, including comments.

FeliCa Networks took the view that the application had been highly effective [Kurita et al. 2008]. From a quality perspective, more errors were found in the early phases of the development than in other similar projects at FeliCa Networks. In total 440 defects were detected in the requirements and the specifications. Of these, 278 were found directly a result of the use of VDM++. Of these, 162 were found by review of the model, whereas 116 were discovered using the VDMTools interpreter with test cases against the executable VDM++ model.

5. OBSERVATIONS

In spite of their successes, verification technology and formal methods have not seen widespread adoption as a routine part of systems development practice, except, arguably, in the development of critical systems in certain domains. Indeed, we expect diffusion of rigorous verification and design technology to take place gradually, and not result in their explicit adoption as a distinct technology [Butterfield 1997].

Previous surveys and our recent review indicate that there have been successes in the application of verification and formal methods to problems of industrial scale and significance, and within industrial settings. Leading hardware developers continue to apply model checking and proof technology. In software, the exploitation of formal techniques has provided some evidence of the potential for applications focused on particular domains, including code verification. Application in the more traditional high-integrity critical applications remains strong, and is largely performed by technical specialists.

In this section, we revisit concerns raised in previous surveys and identify progress, trends, and remaining challenges in the light of more recent projects. We hope that these challenges will be taken up by researchers willing to advance the current state of the art in formal methods.

5.1 Lightweight and Heavyweight Formal Methods

The entry cost for formal methods must seem rather high to the hard-pressed software project manager [Snook and Harrison 2001], although, as noted in Sect. 4.4, the cost of repeated use can decrease dramatically. One of the main difficulties in engineering is the cost-effective choice of what to do and where. No engineer gives the same attention to all the rivets: those below the waterline are singled out; similarly, a formalism need not be applied in full depth to all components of an entire product and through all stages of their development, and that is what we see in practice. The various levels of rigour include the following:

—Best efforts to point out likely generic errors. Examples range from `lint` [Johnson 1978] to Coverity Prevent (a commercial tool developed from the Stanford Checker). The latter performs two sorts of analysis: inter-procedural data-flow analysis, which describes function characteristics with implications for externally observable behaviour; and statistical analysis to detect important trends in the code suggesting anomalies.

—Near-guarantee that all potential errors of a certain class have been flagged. A major exemplar here is the extended static checking of ESC/Java2 [Chalin et al. 2006], a system for automatically detecting at compile-time errors that are normally not detected until run-time, such as: array-bound errors, null dereferences, and race conditions and deadlocks in multi-threaded programs.

—Run-time checking of assertions and other redundant information supplied by the programmer. See [Clarke and Rosenblum 2006] for a historical perspective. Major tools are based on JML [Burdy et al. 2005] and Spec# [Leino 2007].

—Contractual programming, with assertions at major interfaces. Examples include Eiffel [Meyer 1991] and SPARK [Barnes 2003].

—*Lightweight formal methods*. This term has been applied to various forms of application that do not entail such a deep application of the technology. For example, Jones's approach [Jones 1996] favours rigour over full formality, and the production of human-readable proofs that could be formalised if the need arises. Jackson and Wing [Jackson and Wing 1996] suggest a focused application of nonetheless fully formal techniques to partial problems.

All of these deserve targeted research, and the science behind all of them has much in common. It is the duty of the engineer to determine at what places and how far to exploit the science. Our review does not suggest that there is a single strategy for the successful application of verification and formal methods technology. However, a lightweight approach appears to underpin many recent industry applications. Fully formal techniques are used, but are typically focused on specific subsystems and on the verification of particular properties. The success of this approach depends on, and is limited by, the quality of tool support.

The significant developments in tools and tool chains might be expected to affect the forms of deployment of formal techniques in future. Where the achievement of certification is a significant driver, specialist practitioners may be used to construct formal models, formulate conjectures for verification, and guide and interpret the results of semi-automated formal analyses. However, the increasing capability of automated formal analysis makes it possible to have an impact on larger-scale

developments, not necessarily critical, that are driven by the need to reduce time to market, defect rates or costs of testing and maintenance. In such projects, we might expect to see widespread use of enhanced static analysis tools incorporating model checking or proof capabilities.

5.2 Tool Support

The case studies reported in Sect. 4 were predominantly applications of formal methods performed by technical specialists using tools that, with few exceptions, were not felt to be rugged enough for wide-scale application. Almost all surveys in the area point to the importance of producing tools that are well-enough supported to merit commercial application. Particular challenges identified include: support for automated deduction, especially where human interaction with a proof tool is required; common formats for the interchange of models and analyses; and the lack of responsive support for tools users in the case of the many tools offered on a "best efforts" basis. The development of such robust tooling involves addressing practical issues such as providing multi-language support, porting to a variety of platforms, version control, and assistance for co-operative working by multiple engineers on single developments. Several studies listed in Sect. 2 and around a quarter of the projects surveyed in Sect. 3 identify the lack of commercially supported or "ruggedised" tools as an impediment to take-up of formal methods. In spite of the observation that tools are neither necessary nor sufficient for an effective formal methods application [Craigen et al. 1993a], it appears almost inconceivable that an industrial application would now proceed without tools.

The research community has focused considerable effort on the development of new theories to underpin verification, the improvement of tools and, to a limited but increasing extent, tools integration. The work on Mondex originally involved the manual construction of proofs; its more recent revival gives an indication of the improvements in the level of tool support in recent years, whether as a result of improved capability or the underlying Moore's Law increase in processing power. In more recent highlighted projects, and in the new survey, there are examples of robust tools at industrial strength. However, these have still had only limited take-up and there remain many interesting challenges in the underpinning theories of verification as well as in user interaction with verification tools. We must remark that some of the free comments received in the new survey indicate that tools are still not usable, in the words of one respondent, "by mere mortals".

5.3 Increasing Automation

Rushby [Rushby 2000], quoting Norman [Norman 1999], compares verification tools to early radio sets, with "dozens of controls to adjust such arcane features as regeneration and filter bandwidth, and operators were expected to understand the fundamentals of radio reception". A modern radio, on the other hand, is trivial to operate. Model checkers are following this trend, where the goal is to finally become a push-button technology for certain classes of software, such that the trade-off between precision and computational cost of correctness analysis can be controlled by a few simple parameters [Clarke et al. 2008]. Wherever possible, proof tools need to disappear, so that they are fully hidden behind the other tools for analysis, testing, and design automation. Clarke and Wing [Clarke and Wing 1996] anticipate

formal verification technology being applied by developers "with as much ease as compilers". Craigen et al. [Craigen et al. 1993a; 1993b] suggest that tools need to be integral parts of development environments, that the gap between the tools and the standard production process needs to disappear, and that certain analyses could be invoked within existing development environments. There are strong arguments in favour of this approach as a means of moving formal verification technology from innovators to first adopters [Bloomfield and Craigen 1999].

Research is moving towards tools that may be used to provide value-added analyses "under the hood" of existing development environments. The SLAM and SDV experience (Sect. 4), for example, suggests that a targeted approach can yield significant benefits on an existing code base, when the tools are carefully integrated with existing development environments.

Successful take-up of formal verification technology involves "packing specific analyses into easier to use but more restricted components" [Bloomfield and Craigen 1999]. Such automatic analyses must return results that are comprehensible to the user [Arvind et al. 2008]. However, interactions with formal analysis engines must be done at the level of the design language, not the formalism.

Successful integration requires that tools become decoupled components that can be integrated into existing tools for design, programming or static analysis. The integration of multiple verification approaches has been pioneered in development environments such as PROSPER [Dennis et al. 2003] and "plug-in" architectures such as Eclipse have been successfully applied for tools supporting Event-B [RODIN-Project-Members 2007] and VDM [Overture-Core-Team 2007]. Integrations between graphical design notations and the mathematical representations required for formal analysis are increasingly common. For example, the UML to B link [Snook and Butler 2006] allows use of a familiar modelling notation to be coupled to a formal analytic framework, and support for the verification of implementations of systems specified using control law diagrams has been addressed using Z, CSP, and *Circus* [Cavalcanti et al. 2005].

Integration with existing development processes is also significant. The recent survey suggests that model checkers and test generators have significant application in industry. We could speculate that this is because they can be integrated into existing development processes with less upheaval than the adoption of a complete new design or programming framework.

5.4 Cost Effectiveness

Austin and Parkin [Austin and Parkin 1993] note the absence of a convincing body of evidence for the cost-effective use of formal methods in industry as a major barrier to industrial adoption, although they also suggest that it is a problem naturally associated with process change. Craigen et al. [Craigen et al. 1993a] point to the lack of a generally accepted cost model as the reason for this lack of evidence, although past surveys and highlighted projects provide many anecdotes supporting the claim that formal techniques can be used to derive low-defect systems cost effectively. In our survey, only half of the contributions reported the cost consequences, either way, of using formal techniques. The picture is complicated, even where costs are carefully monitored in trial applications of formal methods. The Rockwell Collins work (Sect. 4.4) makes the point that the cost of a one-off formal methods project

is significantly greater than the cost of repeated projects within a domain.

There have been some studies comparing developments with and without the use of formal techniques, such as in the defence sector [Larsen et al. 1996]. Weaknesses were identified in the quantitative claims made for earlier projects [Finney and Fenton 1996], suggesting that strong empirical evidence is needed to encourage adoption. Bloomfield and Craigen [Bloomfield and Craigen 1999] comment that the results of some scientific studies do not scale to engineering problems, and that there has been over-selling and excessive expectation.

Evidence on the technical and cost profiles of commercial applications may not be publicly accessible, but nonetheless, it suggests that pilot studies in verification technology are not always being conducted with a view to gathering information on subsequent stages in technology adoption. Pilot applications of verification technology should be observable and the observations made should be relevant to the needs of those making critical design decisions. For example, the FeliCa networks study (Sect. 4.8) focused on measuring queries against informal requirements documents that were attributable to formal modelling and analysis, because this was seen as a significant feature in a development process where the object was to improve specifications. The Deploy project [Romanovsky 2008; Deploy 2009] explicitly addresses the movement from innovators to early adopters by means of studies measuring the effects of verification tools integrated into existing development processes.

The decision to adopt development technology is risk-based and convincing evidence of the value of formal techniques in identifying defects (with as few false positives as possible) can be at least as powerful as a quantitative cost argument. We would argue for the construction of a strong body of evidence showing the utility of formal techniques and ease of use at least as strongly as we would call for the gathering of more evidence regarding development costs.

6. THE VERIFIED SOFTWARE REPOSITORY

In 2003, Tony Hoare proposed the Verifying Compiler as a Grand Challenge for Computer Science [Hoare 2003]. As the proposal started to gain the support of the community, it became the Grand Challenge in Verified Software [Hoare and Misra 2008] and then the Verified Software Initiative, which was officially launched at the *2008 Conference on Verified Software: Theories, Tools, and Experiments* [Shankar and Woodcock 2008]. The UK effort is in the Grand Challenge in Dependable Systems Evolution (GC6), and current work includes building a Verified Software Repository [Bicarregui et al. 2006].

The Repository will eventually contain hundreds of programs and program modules, amounting to several million lines of code. The code will be accompanied by full or partial specifications, designs, test cases, assertions, evolution histories, and other formal and informal documentation. Each program will have been mechanically checked by one or more tools, and this is expected to be the major activity in the VSI. The eventual suite of verified programs will be selected by the research community as a realistic representative of the wide range of computer applications, including smart cards, embedded software, device routines, modules from a standard class library, an embedded operating system, a compiler for a useful language (possibly Java Card), parts of the verifier itself, a program generator, a communi-

cations protocol (possibly TCP/IP), a desk-top application, parts of a web service (perhaps Apache). We emphasise the academic role of the repository in advancing science, but this does not preclude parts of the repository containing reusable verified components directed towards real-life application domains.

The notion of verification will include the entire spectrum, from avoidance of specific exceptions like buffer overflow, general structural integrity (or crash-proofing), continuity of service, security against intrusion, safety, partial functional correctness, and (at the highest level) total functional correctness [Hoare and Misra 2008]. Similarly, the notion of verification will include the entire spectrum, from unit testing to partial verification through bounded model checking to fully formal proof. To understand exactly what has been achieved, each claim for a specific level of correctness will be accompanied by a clear informal statement of the assumptions and limitations of the proof, and the contribution that it makes to system dependability. The progress of the project will be marked by raising the level of verification for each module in the repository. Since the ultimate goal of the project is scientific, the ultimate level achieved will always be higher than what the normal engineer and customer would accept.

In the following sections we describe five early pilot projects that will be used initially to populate the repository. The verified file-store in Sect. 6.1 is inspired by a space-flight application. FreeRTOS in Sect. 6.2 is a real-time scheduler that is very widely used in embedded systems. The bidding process for the Radio Spectrum Auctions described in Sect. 6.3 has been used with bids ranging from several thousands of dollars to several billions. The cardiac pacemaker in Sect. 6.4 is a real system, and is representative of an important class of medical devices. Finally, Microsoft's hypervisor in Sect. 6.5 is based on one of their future products. The topics of two other pilot projects have been described above: Mondex, in Sect. 4.3, is a smart card for electronic finance; and Tokeneer, in Sect. 4.7, is a security application involving biometrics. These seven pilot projects encompass a wide variety of application areas and each poses some important challenges for verification.

6.1 Verified File Store

Pnueli first suggested the verification of the Linux kernel as a pilot project. Joshi and Holzmann suggested a more modest aim: the verification of the implementation of a subset of the Posix file store interface suitable for flash-memory hardware with strict fault-tolerant requirements to be used by forthcoming NASA missions [Joshi and Holzmann 2007]. The space-flight application requires two important robustness requirements for fault-tolerance: (i) no corruption in the presence of unexpected power-loss; and (ii) recovery from faults specific to flash hardware (*e.g.*, bad blocks, read errors, bit corruption, wear-levelling, *etc.*). In recovery from power loss in particular, the file system is required to be reset-reliable: if an operation is in progress when power is lost, then on reboot, the file system state will be as if the operation either has successfully completed or has never started.

The Posix file-system interface [Josey 2004] was chosen for four reasons: (i) it is a clean, well-defined, and standard interface that has been stable for many years; (ii) the data structures and algorithms required are well understood; (iii) although a small part of an operating system, it is complex enough in terms of reliability guarantees, such as unexpected power-loss, concurrent access, or data

corruption; and (iv) modern information technology is massively dependent on reliable and secure information availability.

An initial subset of the Posix standard has been chosen for the pilot project. There is no support for: (i) file permissions; (ii) hard or symbolic-links; or (iii) entities other than files and directories (*e.g.*, pipes and sockets). Adding support for (i) is not difficult and may be done later, whereas support for (ii) and (iii) is more difficult and might be beyond the scope of the challenge. Existing flash-memory file-systems, such as YAFFS2, do not support these features, since they are not usually needed for embedded systems.

6.2 FreeRTOS

Richard Barry (Wittenstein High Integrity Systems) has proposed the correctness of their open source real-time mini-kernel as a pilot project. FreeRTOS is designed for real-time performance with limited resources, and is accessible, efficient, and popular. It runs on 17 different architectures and is very widely used in many applications. There are over 5,000 downloads per month from SourceForge, making it the repository's 250th most downloaded code (out of 170,000 codes). It is less than 2,500 lines of pointer-rich code. This makes it small, but very interesting.

There are really two challenges here. The first is to analyse the program for structural integrity properties. The second is to make a rational reconstruction of FreeRTOS, starting from an abstract specification, and refining down to working code, with all verification conditions discharged with a high level of automation. These challenges push the current state of the art in both program analysis and refinement of pointer programs.

6.3 Radio Spectrum Auctions

Robert Leese (Smith Institute) has proposed the management of Radio Spectrum Auctions as a pilot project. The radio spectrum is an economically valuable resource, and OfCom, the independent regulator and competition authority for the UK communications industries, holds auctions to sell the rights to transmit over particular wavelengths. The auction is conducted in two phases: the primary stage is a clock auction for price discovery; the supplementary stage is a sealed-bid auction. Both are implemented through a web interface. These auctions are often combinatorial, offering bundles of different wavelengths, which may then also be traded in secondary markets. The underlying auction theory is still being developed, but there are interesting computational problems and challenges to be overcome in getting a trustworthy infrastructure, besides the economic ones.

6.4 Cardiac Pacemaker

Boston Scientific has released into the public domain the system specification for a previous generation pacemaker, and are offering it as a challenge problem. They have released a specification that defines functions and operating characteristics, identifies system environmental performance parameters, and characterises anticipated uses. This challenge has multiple dimensions and levels. Participants may choose to submit a complete version of the pacemaker software, designed to run on specified hardware, they may choose to submit just a formal requirements documents, or anything in between.

McMaster University's Software Quality Research Laboratory is putting in place a certification framework to simulate the concept of licensing. This will enable the Challenge community to explore the concept of licensing evidence and the role of standards in the production of such software. Furthermore, it will provide a more objective basis for comparison between putative solutions to the Challenge.

6.5 Hypervisor

Schulte and Paul initiated work within Microsoft on a hypervisor (a kind of separation kernel), and it has been proposed by Thomas Santen as a challenge project. The European Microsoft Innovation Center is collaborating with German academic partners and the Microsoft Research group for Programming Languages and Methods on the formal verification of the new Microsoft Hypervisor to be released as part of Windows Server 2008. The Hypervisor will allow multiple guest operating systems concurrently on a single hardware platform. By proving the mathematical correctness of the Hypervisor they will control the risks of malicious attack.

7. CONCLUSIONS

In this paper we have presented what is perhaps the most comprehensive review ever made of formal methods application in industry. We see a resurgence of interest in the industrial applications, as shown by the recent emergence of the Verified Software Initiative. In most current applications of formal methods, tools and techniques are being used that have been around for some time, although significant advances in verification technology have yet to filter through to more widespread use. We conclude our paper with some observations on the vitality of formal methods and the maturity of its underlying theory and supporting tools. We discuss how to conduct experiments in this area. Finally, we describe future studies in formal methods, practice and experience.

7.1 Vitality of Formal Methods

Formal methods continue to be a very active research area. There are several specialist journals, including *Formal Aspects of Computing* and *Formal Methods in System Design*, both of which emphasise practical applications as well as theory. The major conferences include the *Formal Methods Symposium* and *Computer-Aided Verification*, both of which have very competitive submissions processes. There was a World Congress in 1999, which will be repeated in 2009. Other important conferences include *Principles of Programming Languages (POPL)*, *Logic in Computer Science (LICS)*, and the *Conference on Automated Deduction (CADE)*. There are very many smaller, specialised conferences and workshops. Some of these are focused on tools and techniques, such as *ABZ*, which covers the Alloy, ASM, B, and Z notations, and the *Refinement Workshop*. Others tackle specific theoretical issues, such as *Integrated Formal Methods*, whilst others cover specific application areas, such as *Formal Methods for Open Object-based Distributed Systems* and *Formal Methods for Human-Computer Interaction*.

The DBLP Computer Science Bibliography [DBLP 2009] contains references to over one million papers, indexed by metadata that is searchable by keywords, and Table IV presents the results of a few searches (dated 9 March 2009). It is interesting to see that there appear to be as many papers published in the general area of

Keyword	Number of papers	Papers/year (2007)
Tools & techniques		
"ACL2"	65	5
"ASM"	364	51
"Alloy"	322	38
"Isabelle"	967	91
"JML"	561	20
"PVS"	109	5
"SPIN model checker"	110	7
"VDM"	303	29
Verification-related keywords		
"Correct"	4,610	387
"Formal methods"	1,473	94
"Model checking"	2,547	288
"Proof"	4,531	313
"Theorem proving"	709	19
"Verification"	10,007	937
Testing and validation		
"Testing"	21,421	1,810
"Validation"	2,902	286
Other areas		
"Compiler"	2,698	128
"Database"	23,333	1,236
"eXtreme programming"	192	5
"Java"	5,217	427
"Object orientation"	7,841	265
"Quantum computation"	2,799	367
"Requirements"	5,275	475
"UML"	2,819	271
"Vision"	8,969	790
Total number of papers in DBLP	1,177,462	118,566

Table IV. DBLP Computer Science Bibliography

verification (23,877 papers, 2,038/year) as there are in testing and validation (24,323 papers, 2,096/year). Verification represents 2% of all papers recorded in DBLP, and verification and validation together account for 4%. At the bottom of the table, we include some keywords from other areas of computer science as a contrast. For example, there are as many papers published in the tools and techniques we have listed as there are in UML; there is twice as much academic interest in JML as in eXtreme Programming; verification and validation each have as many papers as databases, although the latter is a declining area, judging by the rate of publication. Of course, all these figures are rather imprecise, relying on unvalidated metadata.

It is difficult to say precisely how many researchers there are in verification and formal methods, perhaps because these areas have reached a level of maturity that they underpin other areas, with verification being an enabling activity in, say, knowledge acquisition or adaptive systems. It has been estimated that there are 1,000 researchers in verification in the USA [Shankar 2009], with perhaps 300 professors, 200 graduate students, about 250 industrial researchers at places

like Microsoft, Intel, Cisco, IBM, Cadence, Synopsys, and Mentor Graphics, and 50 government researchers in NASA, NSA, NRL, and another 200 or so people doing verification-related work in industry. The UK Engineering and Physical Science Research Council [EPSRC 2009] currently funds about 400 research projects in Software Engineering and the Fundamentals of Computing, with a combined value of £144 million (US$200 million) (it also funds many other topics in computer science, many of which also rely on verification). Of the 144 projects, about 95 mention one of the verification keywords in Table IV, with a combined value of £30 million ($40 million). These projects have 75 different principal investigators with 300 research assistants, making a total with accompanying graduate students of about 500 researchers. Similar rough estimates have been offered for continental Europe (1,000), China (250), the Nordic countries (500), Japan (250), and Australia, Brazil, Canada, New Zealand, Singapore, and South Africa (1,000, collectively). This makes about 4,000 researchers. Of course these figures are very rough, and a more rigorous analysis is needed.

7.2 Maturity of Tools and Advances in Theory

There were "heroic" efforts to use formal methods 20 years ago when few tools were available (to use Bloomfield's phrase [Bloomfield and Craigen 1999]). For example, in the 1980s the application of the Z notation to the IBM CICS transaction processing system was recognised as a major (award-winning) technical achievement [Houston and King 1991], but it is significant that it used only very simple tools: syntax and type-checkers. In the 1990s, the Mondex project (Sect. 4.3) was largely a paper-and-pencil exercise, but it still achieved the highest level of certification. Our evidence is that times have changed: today many people feel that it would be inconceivable not to use some kind of verification tool. Whether they are right or not, there has been a sea-change among verification practitioners about what can be achieved: people seem much more determined to verify industrial problems. This change in attitude, combined with the increase in computing capacity predicted by Moore's Law, and the dramatic advances in research in verification technology (described elsewhere in this issue of *Computing Surveys*) means that the time is right to attempt to make significant advances in the practical application of formal methods and verification in industry. In certain areas, there are collections of mature tools with broadly similar capabilities. For example, the Mondex experiment also shows very similar results from the application of state-based, refinement-oriented techniques and their tools. This suggests that there is scope for convergence and inter-operation between tools.

There has been a move beyond the theory for verification of well-structured subsets of a programming language, to deal with the more potentially undisciplined aspects of programming; for example, pointer swinging [O'Hearn et al. 2001] and concurrency [Vafeiadis and Parkinson 2007]. The increasing popularity of design patterns for disciplined use of these features [Krishnaswami et al. 2009] could support, and be supported by, verification technology.

Satisfiability Modulo Theories (SMT) is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality [Barrett et al. 2008]. The range of theories include integers, real numbers, lists, arrays, and bit vectors. The solvers can handle formulas in

conjunctive normal forms with hundreds of thousands of variables and millions of clauses. They make it possible to apply classical decision procedures to domains where they were previously applicable only in theory [Avigad 2007].

Research in automated theorem proving has made major advances over recent years, particularly in first-order logic, which is expressive enough to specify many problems conveniently. A number of sound and complete calculi have been developed, enabling fully automated systems [TPTP 2009a]. More expressive logics, such as higher-order and modal logics, allow the convenient expression of a wider range of problems than first-order logic, but theorem proving for these logics is less developed. The advancement of these systems has been driven by a large library of standard benchmarks, the *Thousands of Problems for Theorem Provers (TPTP) Problem Library* [TPTP 2009b], and the *CADE ATP System Competition (CASC)*, a yearly competition of first-order systems for many important classes of first-order problems [CADE 2009]. A major opportunity is to turn competition successes into practical application.

7.3 Experimentation

Many projects using formal methods demonstrate early phase benefits, but we were surprised that many respondents to our questionnaire did not know the cost implications of their use of formal methods and verification. Formal methods champions need to be aware of the need to measure costs. Through the Verified Software Initiative, there are many experiments that use material that is realistic to industrial use, and which promises to scale. The experience gained in these experiments contributes to the development of theory, tools, and practice, and it is the tools that will be commercialised to transfer the science and technology to industrial use.

The work reported at Rockwell Collins in Sect. 4.4 is that second and third use of a formal technology and tool chain can lead to order-of-magnitude cost reductions. That is why the first use should always be by scientists, and may be horrifically expensive (we are pushing the boundaries); but even so, the experiment must record costs. This is the experience of the human genome project. The first (composite) human genome cost US$4 billion to sequence. It can now be done for an individual for $4,000. When it gets to $400, the technique will be routinely applied in healthcare.

One of the goals of the VSR should be to set the standard for the well-designed experiment. It should state clearly the hypothesis being tested and address the validity of the experiment as a means of testing the hypothesis. It should try to measure every aspect of the work undertaken, and understand the validity of the measurements being made. But not all experiments in software engineering are set up this way, and we believe that for the Verified Software Repository there should be better appreciation of experimental method. Each experiment must be designed to make the next experiment easier, by leaving a re-usable trail of theory and theorems, by abstracting specifications from a domain of related problems, and by suggesting improvements in the tools used.

7.4 The Future

More details about our survey and its data can be found at [VSR 2009]. We are continuing to collect further data on industrial formal methods projects. Anyone

wishing to contribute should contact any of the authors for further details. We are planning to review the state of practice and experience in formal methods in five, 10, and 15 years' time, as one way of assessing the industrial uptake of formal methods. Perhaps at the end of this we will have hard evidence to support Hoare's vision of a future world in which computer software is always the most reliable component in any system which it controls, and no one can blame the software any more [Hoare 2007].

ACKNOWLEDGMENTS

We thank all the contributors to our survey; the following indicated their willingness to have their names listed: Nikolaj Bjørner, Michael Butler, Gert Caspersen, Mikhail Chupilko, David Crocker, Giuseppe Del Castillo, Asger Eir, Lars-Henrik Eriksson, Alessio Ferrari, Wolfgang Grieskamp, Anthony Hall, Anne Haxthausen, Alexander Kamkin, Rafael Marques, Aad Mathijssen, Steven Miller, Ian Oliver, Alexander Petrenko, Andreas Prinz, Peter Päppenghaus, Thomas Santen, Karl Stroetmann, Nicholas Tudor, Yaroslav Usenko, Eric Verhulst, Michael Whalen, Kirsten Winter, Wolf Zimmermann. We would also like to thank the guest editors, Tony Hoare and Jay Misra, and the anonymous reviewers for their many helpful suggestions for improving our text. Our work has benefited from a previous study carried out for BAE Systems, and we gratefully acknowledge our collaboration with Ana Cavalcanti, Colin O'Halloran, and Nick Tudor, and discussions with Jim Armstrong, Nick Battle, Leo Freitas, Cliff Jones, Steve King, Matt Kaufmann, Joe Kiniry, David Russinoff, Natarajan Shankar, Matt Wilding, and Jeannette Wing. We are grateful to the EU FP7 Integrated Project *Deploy* for valuable discussions. Finally, we thank Microsoft Research Cambridge for supporting the workshop in March 2008 on pilot projects for the Grand Challenge.

REFERENCES

Abrial, J.-R. 1996. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, Cambridge.

Abrial, J.-R. 2007. Formal methods: Theory becoming practice. *JUCS 13,* 5, 619–628.

Allen, R. 1997. A formal approach to software architecture. Ph.D. thesis, School of Computer Science, Carnegie Mellon University. Issued as CMU Technical Report CMU-CS-97-144.

Allen, R. B. and Garlan, D. 1992. A formal approach to software architectures. In *Algorithms, Software, Architecture—Information Processing '92, Volume 1, Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain, 7–11 Sep. 1992*, J. van Leeuwen, Ed. IFIP Transactions, vol. A-12. North-Holland, Amsterdam, 134–141.

Arvind, Dave, N., and Katelman, M. 2008. Getting formal verification into design flow. In *FM 2008: Formal Methods*, J. Cuellar, T. Maibaum, and K. Sere, Eds. Lecture Notes in Computer Science, vol. 5014. Springer-Verlag, Berlin, Heidelberg, 12–32.

Austin, S. and Parkin, G. 1993. Formal methods: A survey. Tech. rep., National Physical Laboratory, Teddington, Middlesex, UK. Mar.

Avigad, J. 2007. Course: Practical decision procedures. `www.andrew.cmu.edu/user/avigad/practical/`. Cited 9 Mar. 2009.

Aydal, E. G., Paige, R. F., and Woodcock, J. 2007. Evaluation of OCL for large-scale modelling: A different view of the Mondex purse. In *MoDELS Workshops*, H. Giese, Ed. Lecture Notes in Computer Science, vol. 5002. Springer, Berlin, Heidelberg, 194–205.

Back, R.-J. and von Wright, J. 1990. Refinement calculus, Part I: Sequential nondeterministic programs. In *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness,*

REX Workshop, Mook, The Netherlands, May 29–Jun. 2, 1989, Proceedings, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds. Lecture Notes in Computer Science, vol. 430. Springer, Berlin, Heidelberg, 42–66.

Badeau, F. and Amelot, A. 2005. Using B as a high level programming language in an industrial project: Roissy VAL. In *ZB*, H. Treharne, S. King, M. C. Henson, and S. A. Schneider, Eds. Lecture Notes in Computer Science, vol. 3455. Springer, Berlin, Heidelberg, 334–354.

Ball, T. and Rajamani, S. K. 2002. The SLAM project: Debugging system software via static analysis. In *POPL, Portland, OR*. ACM, 1–3.

Balser, M., Reif, W., Schellhorn, G., Stenzel, K., and Thums, A. 2000. Formal system development with KIV. In *Fundamental Approaches to Software Engineering, Third International Conference, FASE 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Mar. 25–Apr. 2, 2000, Proceedings*, T. S. E. Maibaum, Ed. Lecture Notes in Computer Science, vol. 1783. Springer, Berlin, Heidelberg, 363–366.

Barnes, J. 2003. *High Integrity Ada: The SPARK Approach to Safety and Security*. Addison-Wesley, Reading, Mass.

Barnes, J., Chapman, R., Johnson, R., Widmaier, J., Cooper, D., and Everett, B. 2006. Engineering the Tokeneer enclave protection system. In *Proceedings of the 1st International Symposium on Secure Software Engineering, Arlington, VA*. IEEE.

Barrett, C., Sebastiani, R., Seshia, S. A., and Tinelli, C. 2008. Satisfiability modulo theories. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsch, Eds. IOS Press, Amsterdam, Chapter 12, 737–797.

Barrett, G. 1987. Formal methods applied to a floating-point number system. Technical Monograph PRG-58, Oxford University Computing Laboratory.

Barrett, G. 1989. Formal methods applied to a floating-point number system. *IEEE Trans. Software Eng. 15,* 5, 611–621.

Barrett, G. 1990. Verifying the Transputer. In *Transputer Research and Applications, Proceedings of the 1st Conference of the North American Transputer Users Group*, G. S. Stiles, Ed. IOS, Amsterdam, 17–24.

Behm, P., Benoit, P., Faivre, A., and Meynadier, J.-M. 1999. Météor: A successful application of B in a large project. In *World Congress on Formal Methods*, J. M. Wing, J. Woodcock, and J. Davies, Eds. Lecture Notes in Computer Science, vol. 1708. Springer, Berlin, Heidelberg, 369–387.

Berry, G. 2008. Synchronous design and verification of critical embedded systems using SCADE and Esterel. In *Formal Methods for Industrial Critical Systems*. Lecture Notes in Computer Science, vol. 4916. Springer, Berlin, Heidelberg.

Bicarregui, J., Hoare, C. A. R., and Woodcock, J. C. P. 2006. The Verified Software Repository: a step towards the verifying compiler. *Formal Asp. Comput. 18,* 2, 143–151.

Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X. 2003. A static analyzer for large safety-critical software. In *PLDI, San Diego*. ACM, 196–207.

Bloomfield, R. and Craigen, D. 1999. Formal methods diffusion: Past lessons and future prospects. Tech. Rep. D/167/6101, Adelard, Coborn House, 3 Coborn Road, London E3 2DA, UK. Dec.

Börger, E. and Stärk, R. 2003. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, Berlin, Heidelberg.

Bowen, J. and Stavridou, V. 1993. Safety-critical systems, formal methods and standards. *Software Engineering Journal 8,* 4, 189–209.

Bowen, J. P. and Hinchey, M. G. 1995. Ten commandments of formal methods. *IEEE Computer 28,* 4 (Apr.), 56–62.

Bowen, J. P. and Hinchey, M. G. 2006. Ten commandments of formal methods... ten years later. *IEEE Computer 39,* 1 (Jan.), 40–48.

Burdy, L., Cheon, Y., Cok, D. R., Ernst, M. D., Kiniry, J. R., Leavens, G. T., Leino, K. R. M., and Poll, E. 2005. An overview of JML tools and applications. *STTT 7,* 3, 212–232.

Bush, W. R., Pincus, J. D., and Sielaff, D. J. 2000. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper. 30,* 7, 775–802.

Butler, M. and Yadav, D. 2008. An incremental development of the Mondex system in Event-B. *Formal Asp. Comput. 20,* 1, 61–77.

Butterfield, A. 1997. Introducing formal methods to existing processes. In *IEE Colloquium on Industrial Use of Formal Methods, London, 23 May 1997*. IET, 7–10.

CADE. 2009. The cade atp system competition: The world championship for 1st order automated theorem proving. `www.cs.miami.edu/~tptp/CASC/`. Cited 9 Mar. 2009.

Cavalcanti, A., Clayton, P., and O'Halloran, C. 2005. Control law diagrams in *circus*. In *FM 2005: Formal Methods*, J. Fitzgerald, I. J. Hayes, and A. Tarlecki, Eds. Lecture Notes in Computer Science, vol. 3582. Springer-Verlag, Berlin, Heidelberg, 253–268.

CCRA. 2006. Common criteria for information technology security evaluation. Part 1: Introduction and general model. Tech. Rep. CCMB-2006-09-001, Version 3.1, Revision 1, Common Criteria Recognition Agreement. Sep.

Chalin, P., Kiniry, J. R., Leavens, G. T., and Poll, E. 2006. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, Nov. 1–4, 2005, Revised Lectures*, F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, Eds. Lecture Notes in Computer Science, vol. 4111. Springer, Berlin, Heidelberg, 342–363.

Chapman, R. 2008. Tokeneer ID Station Overview and Reader's Guide. Tech. Rep. S.P1229.81.8, Issue 1.0, Praxis High Integrity Systems.

Clarke, E. M., Gupta, A., Jain, H., and Veith, H. 2008. Model checking: Back and forth between hardware and software. In *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, B. Meyer and J. Woodcock, Eds. Lecture Notes in Computer Science, vol. 4171. Springer, Berlin, Heidelberg, 251–255.

Clarke, E. M. and Wing, J. M. 1996. Formal methods: State of the art and future directions. *ACM Computing Surveys 28,* 4, 626–643.

Clarke, L. A. and Rosenblum, D. S. 2006. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes 31,* 3, 25–37.

Craigen, D., Gerhart, S., and Ralston, T. 1993a. *An International Survey of Industrial Applications of Formal Methods.* Vol. 1 Purpose, Approach, Analysis and Conclusions. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD.

Craigen, D., Gerhart, S., and Ralston, T. 1993b. *An International Survey of Industrial Applications of Formal Methods.* Vol. 2 Case Studies. U.S. Department of Commerce, Technology Administration, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD.

Cuadrado, J. 1994. Teach formal methods. *Byte 19,* 12 (Dec.), 292.

Das, M., Lerner, S., and Seigle, M. 2002. ESP: Path-sensitive program verification in polynomial time. In *PLDI, Berlin*. ACM, 57–68.

DBLP. 2009. Computer Science bibliography. `www.informatik.uni-trier.de/~ley/db/`. Cited 9 Mar. 2009.

Dennis, L. A., Collins, G., Norrish, M., Boulton, R. J., Slind, K., and Melham, T. F. 2003. The PROSPER toolkit. *International Journal on Software Tools for Technology Transfer 4,* 2 (Feb.), 189–210.

Deploy. 2009. `www.deploy-project.eu`. Cited 9 Mar. 2009.

Dijkstra, E. W. 1975. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM 18,* 8, 453–457.

EPSRC. 2009. Engineering and Physical Sciences Research Council. `www.epsrc.ac.uk`. Cited 9 Mar. 2009.

Evans, D. and Larochelle, D. 2002. Improving security using extensible lightweight static analysis. *IEEE Software 19*, 42–52.

Finney, K. and Fenton, N. 1996. Evaluating the effectiveness of Z: The claims made about CICS and where we go from here. *Journal of Systems and Software 35*, 209–216.

Fitzgerald, J., Larsen, P. G., and Sahara, S. 2008. VDMTools: Advances in support for formal modeling in VDM. *Sigplan Notices 43,* 2 (Feb.), 3–11.

Floyd, R. 1967. Assigning meanings to programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, Providence, Rhode Island*. American Mathematical Society, 19–32.

Freitas, L. and Woodcock, J. 2008. Mechanising Mondex with Z/Eves. *Formal Asp. Comput. 20,* 1, 117–139.

Gaudel, M.-C. 1995. Testing can be formal, too. In *TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22–26, 1995, Proceedings*, P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, Eds. Lecture Notes in Computer Science, vol. 915. Springer, Berlin, Heidelberg, 82–96.

George, C. and Haxthausen, A. E. 2008. Specification, proof, and model checking of the Mondex electronic purse using RAISE. *Formal Asp. Comput. 20,* 1, 101–116.

George, V. and Vaughn, R. 2003. Application of lightweight formal methods in requirement engineering. *Crosstalk: The Journal of Defense Software Engineering*.

Ghose, A. 2000. Formal methods for requirements engineering. In *2000 International Symposium on Multimedia Software Engineering (ISMSE 2000), Taipei, Taiwan, 11–13 Dec. 2000*. IEEE Computer Society, 13–16.

Gibbons, J. 1993. Formal methods: Why should I care? The development of the T800 Transputer floating-point unit. In *Proceedings of the 13th New Zealand Computer Society Conference*, J. Hosking, Ed. New Zealand Computer Society, Auckland, 207–217.

Gibbs, W. W. 1994. Software's chronic crisis. *Scientific American 271,* 3 (Sep.), 72–81.

Glass, R. L. 1996. Formal methods are a surrogate for a more serious software concern. *IEEE Computer 29,* 4 (Apr.), 19.

Goldsmith, M., Cox, A., and Barrett, G. 1987. An algebraic transformation system for Occam programs. In *STACS*, F.-J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, Eds. Lecture Notes in Computer Science, vol. 247. Springer, Berlin, Heidelberg, 481.

Greve, D. and Wilding, M. 2002. Evaluatable, high-assurance microprocessors. In *NSA High-Confidence Systems and Software Conference (HCSS), Linthicum, MD, Mar. 6–8*. NSA.

Guiho, G. D. and Hennebert, C. 1990. SACEM software validation (experience report). In *ICSE: Proceedings 12th International Conference on Software Engineering, Nice*. IEEE Computer Society, 186–191.

Hall, A. 1990. Seven myths of formal methods. *IEEE Software 7,* 5 (Sep.), 11–19.

Hall, A. and Chapman, R. 2002. Correctness by construction: Developing a commercial secure system. *IEEE Software 19,* 1, 18–25.

Haneberg, D., Schellhorn, G., Grandy, H., and Reif, W. 2008. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. *Formal Asp. Comput. 20,* 1, 41–59.

Hennebert, C. and Guiho, G. D. 1993. SACEM: A fault tolerant system for train speed control. In *FTCS, Toulouse*. IEEE Computer Society, 624–628.

Hierons, R. M., Bowen, J. P., and Harman, M., Eds. 2008. *Formal Methods and Testing An Outcome of the FORTEST Network. Revised Selected Papers*. Lecture Notes in Computer Science, vol. 4949. Springer, Berlin, Heidelberg.

Hinchey, M. G. and Bowen, J. P. 1995. *Applications of Formal Methods*. Prentice Hall, Englewood Cliffs NJ.

Hinchey, M. G. and Bowen, J. P. 1996. To formalize or not to formalize? *IEEE Computer 29,* 4 (Apr.), 18–19.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM 12,* 10 (Oct.), 576–581.

Hoare, C. A. R. 1972. Proof of correctness of data representations. *Acta Inf. 1*, 271–281.

Hoare, C. A. R. 1985. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ.

Hoare, C. A. R. 2002a. Assertions in modern software engineering practice. In *26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life: Development and Redevelopment, Oxford, 26–29 Aug. 2002, Proceedings*. IEEE Computer Society, 459–462.

Hoare, C. A. R. 2003. The verifying compiler: A Grand Challenge for computing research. *J. ACM 50,* 1, 63–69.

Hoare, T. 2002b. Assert early and assert often: Practical hints on effective asserting. Presentation at Microsoft Techfest.

Hoare, T. 2007. The ideal of program correctness: *Third Computer Journal Lecture. Computer Journal 50,* 3, 254–260.

Hoare, T. and Misra, J. 2008. Verified software: Theories, tools, and experiments: Vision of a Grand Challenge project. In *Verified Software: Theories, Tools, and Experiments. First IFIP TC2/EG2.3 Conference, Zurich, Oct. 2005*, B. Meyer and J. Woodcock, Eds. Lecture Notes in Computer Science, vol. 4171. Springer, Berlin, Heidelberg, 1–18.

Holzmann, G. J. 2004. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Reading, Mass.

Hommersom, A., Groot, P., Lucas, P. J. F., Balser, M., and Schmitt, J. 2007. Verification of medical guidelines using background knowledge in task networks. *IEEE Trans. Knowl. Data Eng. 19,* 6, 832–846.

Houston, I. and King, S. 1991. Experiences and results from the use of Z in IBM. In *VDM '91: Formal Software Development Methods*. Lecture Notes in Computer Science, vol. 551. Springer, Berlin, Heidelberg, 588–595.

IEC. 1997. Functional safety of electrical/electronic/programmable electronic safety-related systems. Tech. Rep. IEC 61508, International Electrotechnical Commission.

IEEE. 1985. *Standard for Binary Floating-Point Arithmetic*, Standard 754-1985 ed. ANSI/IEEE.

Inmos Ltd. 1988a. *Occam2 Reference Manual*. Prentice Hall, Englewood Cliffs, NJ.

Inmos Ltd. 1988b. *Transputer Reference Manual*. Prentice Hall, Englewood Cliffs, NJ.

ITSEC. 1991. Information technology security evaluation criteria (ITSEC): Preliminary harmonised criteria. Tech. Rep. Document COM(90) 314, Version 1.2, Commission of the European Communities. Jun.

Jackson, D. and Wing, J. 1996. Lightweight formal methods. *IEEE Computer 29,* 4 (Apr.), 22–23.

Johnson, S. 1978. `lint`, a C program checker, Unix Programmers Manual. Tech. Rep. 65, AT&T Bell Laboratories.

Jones, C. 1996. A rigorous approach to formal methods. *IEEE Computer 29,* 4 (Apr.), 20–21.

Jones, C. B. 1990. *Systematic Software Development Using VDM*, 2nd ed. Prentice-Hall International, Englewood Cliffs, NJ.

Jones, C. B. and Pierce, K. G. 2007. What can the π-calculus tell us about the Mondex purse system? In *12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007), Auckland, New Zealand, 10–14 Jul. 2007*. IEEE Computer Society, 300–306.

Jones, G. and Goldsmith, M. 1988. *Programming in Occam2*. Prentice Hall, Englewood Cliffs, NJ.

Josey, A. 2004. *The Single* Unix *Specification Version 3*. Open Group, San Francisco, CA. ISBN: 193162447X.

Joshi, R. and Holzmann, G. J. 2007. A mini challenge: Build a verifiable filesystem. *Formal Asp. Comput. 19,* 2, 269–272.

Kars, P. 1997. The application of PROMELA and SPIN in the BOS project. In *The SPIN Verification System: The Second Workshop on the SPIN Verification System. Proceedings of a DIMACS Workshop, Aug. 5, 1996*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 33. Rutgers University, Piscataway, NJ, 51–63.

Krishnaswami, N. R., Aldrich, J., Birkedal, L., Svendsen, K., and Buisse, A. 2009. Design patterns in separation logic. In *Proceedings of TLDI'08: 2008 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, January 24, 2009*, A. Kennedy and A. Ahmed, Eds. ACM, 105–116.

Kuhlmann, M. and Gogolla, M. 2008. Modeling and validating Mondex scenarios described in UML and OCL with USE. *Formal Asp. Comput. 20,* 1, 79–100.

Kurita, T., Chiba, M., and Nakatsugawa, Y. 2008. Application of a formal specification language in the development of the "Mobile FeliCa" IC chip firmware for embedding in mobile phones. In *FM 2008: Formal Methods*, J. Cuellar, T. Maibaum, and K. Sere, Eds. Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, 425–429.

Larsen, P. G. and Fitzgerald, J. 2007. Recent industrial applications of VDM in Japan. In *FACS 2007 Christmas Workshop: Formal Methods in Industry*, P. Boca, J. Bowen, and P. Larsen, Eds. BCS, eWIC, London.

Larsen, P. G., Fitzgerald, J., and Brookes, T. 1996. Applying formal specification in industry. *IEEE Software 13,* 3 (May), 48–56.

Larus, J. R., Ball, T., Das, M., DeLine, R., Fähndrich, M., Pincus, J. D., Rajamani, S. K., and Venkatapathy, R. 2004. Righting software. *IEEE Software 21,* 3, 92–100.

Leino, K. R. M. 2007. Specifying and verifying programs in Spec#. In *Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference, PSI 2006, Novosibirsk, Russia, Jun. 27–30, 2006. Revised Papers*, I. Virbitskaite and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 4378. Springer, Berlin, Heidelberg, 20.

May, D., Barrett, G., and Shepherd, D. 1992. Designing chips that work. *Philosophical Transactions of the Royal Society A 339*, 3–19.

Meyer, B. 1991. In *Advances in Object-Oriented Software Engineering*, D. Mandrioli and B. Meyer, Eds. Prentice Hall, Englewood Cliffs, NJ, Chapter Design by Contract, 1–50.

Miller, S. and Srivas, M. 1995. Formal verification of the AAMP5 microprocessor. In *Workshop on Industrial-Strength Formal Specification Techniques (WIFT95), Boca Raton, Florida, Apr. 5–8*. IEEE Computer Society.

Miller, S. P. 1998. The industrial use of formal methods: Was Darwin right? In *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, Boca Raton, FL*. IEEE Computer Society, 74–82.

Miller, S. P., Greve, D. A., and Srivas, M. K. 1996. Formal verification of the AAMP5 and AAMP-FV microcode. In *Third AMAST Workshop on Real-Time Systems, Salt Lake City, Utah, Mar. 6–8, 1996*.

Morgan, C. 1988. The specification statement. *ACM Trans. Program. Lang. Syst. 10,* 3, 403–419.

Morris, J. M. 1987. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Comput. Program. 9,* 3, 287–306.

NASA. 1997. Formal methods, specification and verification guidebook for the verification of software and computer systems. vol II: A practitioner's companion. Tech. Rep. NASA-GB-001-97, Washington, DC. May.

NASA. 1998. Formal methods, specification and verification guidebook for the verification of software and computer systems. vol I: Planning and technology insertion. Tech. Rep. NASA/TP-98-208193, Washington, DC. Dec.

Norman, D. A. 1999. *The Invisible Computer: Why Good Products Can Fail, the Personal Computer Is So Complex, and Information Appliances Are the Solution*. The MIT Press.

O'Hearn, P. W., Reynolds, J. C., and Yang, H. 2001. Local reasoning about programs that alter data structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings*, L. Fribourg, Ed. Lecture Notes in Computer Science, vol. 2142. Springer, Berlin, Heidelberg, 1–19.

Overture-Core-Team. 2007. Overture web site. `www.overturetool.org`. Cited 9 Mar. 2009.

Owre, S., Rushby, J. M., and Shankar, N. 1992. PVS: A prototype verification system. In *CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, Jun.*

15–18 1992. Lecture Notes in Computer Science, vol. 607. Springer, Berlin, Heidelberg, 748–752.

RAISE Language Group. 1992. *The RAISE Specification Language*. The BCS Practitioners Series. Prentice-Hall.

RAISE Method Group. 1995. *The RAISE Development Method*. The BCS Practitioners Series. Prentice-Hall International.

Ramananandro, T. 2008. Mondex, an electronic purse: Specification and refinement checks with the Alloy model-finding method. *Formal Asp. Comput. 20,* 1, 21–39.

Riazanov, A. and Voronkov, A. 2002. The design and implementation of VAMPIRE. *AI Commun. 15,* 2–3, 91–110.

RODIN-Project-Members. 2007. RODIN web site. `rodin.cs.ncl.ac.uk/`. Cited 9 Mar. 2009.

Romanovsky, A. 2008. DEPLOY: Industrial deployment of advanced system engineering methods for high productivity and dependability. *ERCIM News 74*, 54–55.

Roscoe, A. W. and Hoare, C. A. R. 1988. The laws of Occam programming. *Theoretical Computer Science 60*, 177–229.

Rushby, J. 1993. Formal methods and the certification of critical systems. Tech. Rep. CSL-93-7, Computer Science Laboratory, Menlo Park, CA. Dec.

Rushby, J. 2000. Disappearing formal methods. In *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposium on HASE 2000*. IEEE.

Saiedian, H. 1996. An invitation to formal methods. *IEEE Computer 29,* 4 (Apr.), 16–30.

Shankar, N. 2009. Personal communication.

Shankar, N. and Woodcock, J., Eds. 2008. *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, Oct. 6–9, 2008. Proceedings*. Lecture Notes in Computer Science, vol. 5295. Springer, Berlin, Heidelberg.

Shepherd, D. 1988. The role of Occam in the design of the IMS T800. In *Communicating Process Architectures*. Prentice Hall.

Shepherd, D. and Wilson, G. 1989. Making chips that work. *New Scientist 1664*, 39–42.

Snook, C. and Butler, M. 2006. UML-B: Formal modeling and design aided by UML. *ACM Trans. Softw. Eng. Methodol. 15,* 1, 92–122.

Snook, C. and Harrison, R. 2001. Practitioners' views on the use of formal methods: An industrial survey by structured interview. *Information and Software Technology 43*, 275–283.

Spinellis, D. 2008. A look at zero-defect code. `www.spinellis.gr/blog/20081018/`. Cited 9 Mar. 2009.

Spivey, J. M. 1989. *The Z Notation: a Reference Manual*. International Series in Computer Science. Prentice Hall.

Srivas, M. and Miller, S. 1995. Applying formal verification to a commercial microprocessor. In *IFIP Conference on Hardware Description Languages and their Applications (CHDL'95), Makuhari, Chiba, Japan*.

Stepney, S., Cooper, D., and Woodcock, J. 2000. An electronic purse: Specification, refinement, and proof. Technical Monograph PRG-126, Oxford University Computing Laboratory. Jul.

Thomas, M. 1992. The industrial use of formal methods. *Microprocessors and Microsystems 17,* 1 (Jan.), 31–36.

Tiwari, A., Shankar, N., and Rushby, J. 2003. Invisible formal methods for embedded control systems. *Proceedings of the IEEE 91,* 1 (Jan), 29–39.

Tokeneer. 2009. `www.adacore.com/tokeneer`. Cited 9 Mar. 2009.

TPTP. 2009a. Entrants' system descriptions. `www.cs.miami.edu/~tptp/CASC/J2/SystemDescriptions.html`. Cited 9 Mar. 2009.

TPTP. 2009b. The TPTP problem library for automated theorem proving. `www.cs.miami.edu/~tptp/`. Cited 9 Mar. 2009.

Tretmans, J., Wijbrans, K., and Chaudron, M. 2001. Software engineering with formal methods: The development of a storm surge barrier control system revisiting seven myths of formal methods. *Form. Methods Syst. Des. 19,* 2, 195–215.

Vafeiadis, V. and Parkinson, M. J. 2007. A marriage of rely/guarantee and separation logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3–8, 2007, Proceedings*, L. Caires and V. T. Vasconcelos, Eds. Lecture Notes in Computer Science, vol. 4703. Springer, Berlin, Heidelberg, 256–271.

van Lamsweerde, A. 2003. From system goals to software architecture. In *SFM*. 25–43.

VSR. 2009. Verified Software Repository. `vsr.sourceforge.net/fmsurvey.htm`. Cited 9 Mar. 2009.

Ward, M. P. and Bennett, K. H. 1995. Formal methods to aid the evolution of software. *International Journal of Software Engineering and Knowledge Engineering 5*, 25–47.

Wijbrans, K., Buve, F., Rijkers, R., and Geurts, W. 2008. Software engineering with formal methods: Experiences with the development of a storm surge barrier control system. In *FM2008: Formal Methods*, J. Cuellar, T. Maibaum, and K. Sere, Eds. Vol. 5014. Springer-Verlag, Berlin, Heidelberg, 419–424.

Wilding, M., Greve, D., and Hardin, D. 2001. Efficient simulation of formal processor models. *Formal Methods in Systems Design 18,* 3 (May), 233–248.

Wing, J. M. 1990. A specifier's introduction to formal methods. *IEEE Computer 23,* 9, 8–24.

Woodcock, J. and Davies, J. 1996. *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science. Prentice Hall.

Woodcock, J., Stepney, S., Cooper, D., Clark, J. A., and Jacob, J. 2008. The certification of the Mondex electronic purse to ITSEC Level E6. *Formal Aspects of Computing 20,* 1, 5–19.

Younger, E. J., Luo, Z., Bennett, K. H., and Bull, T. M. 1996. Reverse engineering concurrent programs using formal modelling and analysis. In *1996 International Conference on Software Maintenance (ICSM '96), 4–8 Nov. 1996, Monterey, CA, Proceedings*. IEEE Computer Society, 255–264.

ACRONYM LIST

ASM	Abstract State Machine
CA	Certification Authority
CADE	Conference on Automated Deduction
CASE	Computer Aided Software Engineering
CENELEC	The European Committee for Electrotechnical Standardization
CICS	Customer Information Control System
CSP	Communicating Sequential Processes
EAL	Evaluation Assurance Level
EMIC	European Microsoft Innovation Center
FDR	Failure Divergence Refinement
HCI	Human Computer Interface
IEC	International Electrotechnical Commission
ITSEC	Information Technology Security Evaluation Criteria
KIV	Karlsruhe Interactive Verifier
KLOC	Kilo-Lines of Code
MBD	Model Based Development
NSA	National Security Agency
OCL	Object Constraint Language
PROSPER	Proof and Specification Assisted Design Environments
RODIN	Rigorous Open Development Environment for Complex Systems
PVS	Prototype Verification System
RTE	Real Time Executive
RTOS	Real Time Operating System
SCADE	Safety Critical Application Development Environment
SDV	Static Driver Verifier
SIL	Safety Integrity Level
UML	Unified Modelling Language
VDM	Vienna Development Method
VSI	Verified Software Initiative