Vision Paper: Remodelling Software Systems – the 2020 Grand Challenge for Software Engineering

Anthony J H Simons[1] and W Michael L Holcombe[2],

[1, 2] Department of Computer Science, University of Sheffield,
211 Portobello, Sheffield S1 4DP, UK
[1] A.J.Simons@shef.ac.uk, [2] M.Holcombe@dcs.shef.ac.uk

Abstract. In 2002, the UK computer science research community set itself a number of Grand Challenges to focus efforts in the discipline. Surprisingly, software engineering only featured incidentally in the challenges that were eventually tabled. Since then, we have tried to develop what we believe is a worthier and more comprehensive challenge for software engineering: to deliver and continuously upgrade large-scale software systems, for which the end-user requirements are constantly evolving. The technology to do this requires major scientific advances, such as model-driven re-engineering to adapt and cannibalise systems at a high level of abstraction, and proven model transformation methods for converting evolving designs into code. Critically, systems must evolve during live operation, without compromising functionality or invalidating business data. We break this Grand Challenge down into six sub-areas needing substantial scientific progress, and set a target competition for 2020.

Keywords: Model-driven engineering, end-user programming, self-verification and testing, uninterrupted service

1 Introduction

If you were to ask the well-informed man in the street about the main challenges facing software engineering, he would still probably reply along the lines that large government-funded software projects are frequently behind schedule, are sometimes never delivered and waste billions of taxpayer's dollars. He might also remark on the ubiquitous growth in software systems, the escalation into the Cloud, and express apprehension about whether these systems work as expected, whether his personal data is secure anymore, and ask when the first socio-economic meltdown caused by software failure is likely to occur. As an afterthought, he might add that the latest version of his desktop office software has an incomprehensible user interface and his organisation's adoption of the *TotalControl*™ centralised management system has made his life hell, requiring him constantly to undertake reporting tasks that should be handled by his secretariat[1].

[1] We dare not reveal the real name of this well-known ERP system.

1.1 Software Engineering, 2000-2010

The uncomfortable truth is that we are still not very good at software engineering. *The Guardian* reported that the UK government had wasted £2bn (nearly $4bn) on cancelled software projects between 2000-2008 [1], while *The Independent* calculated that by 2010, after cut-backs triggered by the financial crisis, the bill was more like £26bn ($50bn) including cost overruns [2]. A 2005 article in *IEEE Spectrum* catalogued a long list of avoidable software failures [3] in public service and private enterprise, estimating that the US government had wasted between $25-75bn over five years on failed projects. A survey by the British Computer Society studied 214 software projects in the European Union from 1995-2005 and found that only 12.5% could be judged successful by their own criteria and €142bn ($169bn) was written off in 2004 alone [4]. Each of these studies cite lack of diligence over the requirements as a significant cause of failure, breaking this down into:

- inability to deal with change requests in the requirements;
- failure to communicate between the developers and stakeholders;
- no clear requirements definitions.

Business reasons for project cancellation included that the business strategy had been superseded, the business processes had changed or the costs and benefits of the system had been poorly explained. The customer-facing engineers were often ill chosen, such that stakeholders could not sign off design work that seemed divorced from their own understanding of their business [4]. While problems were found at every technical level, from the wrong choice of architecture to unskilled coding and inadequate testing, more disasters were caused by managerial inexperience. Contracts were negotiated by bid and counter-bid, rather by evidence-based estimation using metrics. Poor training, contractual and financial management were cited, along with failure to understand or manage risk. Projects were prolonged out of bravado, after independent analysts had determined that they could never recoup their costs [3].

1.2 Grand Challenges in Computing Research, 2002-2010

In the early 2000s, with the rise of specialised computational techniques in other disciplines, Computer Science found itself under threat of being marginalized in the UK as an adjunct to every other discipline. Responding to this, the UK Computing Research Committee (UKCRC), led by professors Sir Tony Hoare and Robin Milner, issued a tender for proposals constituting Grand Challenges in Computing Research. The suggestions received fell into four broad categories [5]:

- formal methods and verification driven software development
- concurrent and mobile architecture for ubiquitous computing
- human-oriented smart digital services, lifestyle and education
- nature of mind and intelligence, fusion of *in vivo* and *in silico*

Eventually, the submissions were marshalled into eight Grand Challenges [6], although the GC process is intended to be ongoing and in 2010 the UKCRC issued a call for new proposals. What struck us as disappointing at the time was the peripheral

way in which software engineering was treated as an adjunct of the interests of the formal methods community. The two challenges most related to software engineering were: *Ubiquitous Computing*, whose focus is on the emergence of the "smart cloud" of small, personal computing devices and aims to investigate the theory, practice and social impact of mobile, concurrent networked devices; and *Dependable Systems Evolution*, which seeks to build a self-verifying compiler and develop a repository of code whose behaviour is formally proven to be correct. Now, these clearly target the core research interests of the calculus and refinement communities, respectively, and are certainly valid concerns in their own right; but to our mind, they do not address the real underlying problems in software engineering.

1.3 Towards a Grand Challenge in Software Engineering?

The 2010 call for new Grand Challenges attracted a software engineering-related submission that hit the nail more squarely on the head [6]: *Hatching a Phoenix: Reclaiming Success from Software Failure*, a challenge to use open source and agile development to build generic versions of failed government software projects. This questioned how software is commissioned and delivered, targeting the combination of secrecy and vested interest that was slated in [2]. The UK government is also waking up to the trends of the last 10 years, highlighting open source, small projects, agile teams and a "Skunkworks" for SMEs to participate in rapid innovation, in its 2011 plan for future ICT procurement [7].

A commendable aspect of the *Phoenix* proposal is that it sets a target to achieve within a fixed timeframe: the reconstruction of the failed C-NOMIS system for tracking offenders through the justice and rehabilitation system. Achieving this would be a major coup. The original GC call cited the 1960s NASA project to "place a man on the moon within 10 years" as an archetype of the kind of submission they expected, an ambitious goal that would drive innovation in science and engineering, through concerted collaborative effort. This goal-driven aspect is one of the maturity criteria for GC submissions [6], and is only possible once all the facets of the problem are understood well enough for a concerted effort to be possible.

We believe that it is time to formulate a comprehensive Grand Challenge for Software Engineering. It is not just the procurement policy that is in question, nor is the correctness of compilers the main issue. The heart of the problem has to do with the fact that modern software systems are the most complex artefacts ever engineered by humans and no one mind, or group of minds, can encompass the whole. This should really make the agile movement think twice: if the next incremental requirement triggers the cascading refactoring of the whole system, what are the consequences of not comprehending the whole system? If abstraction is the answer, why should we expect good abstract design to emerge by accident [8]? But even this isn't the main showstopper for software engineering. The main problem, as highlighted in all the earlier studies [1-4], is not being able to satisfy stakeholder requirements. Partly this comes from the rush to embrace technology at the expense of understanding the ergonomics of the workplace[2]; but mostly from the rapid pace of

[2] Remember that famous ERP system mentioned in the introduction?

change, the fact that, given current methods of software production, developers cannot hope to keep up with evolving stakeholder requirements.

2 Remodelling Software Systems

The solution to this problem must include a paradigm shift from perpetual tinkering with low-level program code to the rapid, mechanised creation of software systems from high-level abstract models. Currently, we are enjoying a historical period where open source and agile development is an effective strategy for breaking the old monopoly of traditional software providers and slow, inflexible processes. But there is a size and comprehension limit at which upgrading software systems by continually editing the code-base ceases to be productive and costs more to maintain (in man-hours) than the economic benefit accrued by having the system [3]. Agile programming must recognise that it has a use-by date.

Now, stating that model-driven engineering must be part of the solution will come as no surprise to the MoDELS conference and wider MDE community. However, the purpose of this vision paper and call to action is to show just how far we still have to go. There are ideas we have embraced, but whose value is misunderstood. There are false paths we have followed. There are practical aspects we have failed to consider. There are large gaps in our understanding that still remain to be filled. The following aims to set out a research agenda for the next decade. First, we explain the kind of challenge that we expect model-driven engineering to address.

2.1 The Software Engineering Grand Challenge for 2020

We envisage a contest, to which all interested parties may bring their best tools and methods, to be evaluated competitively in a supportive environment, similar to the DARPA contest TREC [9]. Entrants will be given only *one week* in which to design and build, from scratch, a substantial enterprise-level information system. The stakeholders will present their requirements at the start of the week and will be available for each team to interview, on an equal basis, throughout the week. At the end of day 3, it is expected that the system requirements will be signed off, implying that the stakeholders will fully understand and agree with the models produced by the developers (c.f. [4]). By the close of day 4, it is expected that a prototype version of the system will be implemented and populated with real business data.

On day 5, the stakeholders will present a revised set of requirements. These will not just contain extra features, but will include at least one requirement that has a major impact on the system architecture, indicating a change of direction for the project, such as might be forced by the need to interoperate with some external system, or by legal changes governing the way the business is regulated. The teams will have until the end of day 6 to modify the original system to meet the new specification. The changes forced by the new requirements are expected to be pervasive, in the sense that they could not be achieved within the timescale simply by intervening at the source code-level, but rather only by regenerating the system from substantially customised models.

On day 7, the systems will be evaluated for compliance to the requirements, for correct, robust and timely performance and for usability by the intended stakeholders. The evaluation panels will consist of referees (for compliance testing), stakeholders (for usability testing) and developers (for robustness testing). The referees will be the final arbiters, shared across different team evaluations, to ensure consistent, fair treatment. Eventually, the winners will be ranked.

There are many possible variations to this competition; for example, you could start with a known prior information system, which is then heavily customised for the challenge system. You could also have more than one evolution step in which new requirements were introduced during the week.

2.2 The Vision for Software Systems Remodelling

We call this kind of rapid system development and customisation *remodelling*, since the focus is on reusing and adapting high-level models of software systems, rather than on *refactoring* programs, as at present. To reach the point at which this becomes a feasible proposition for mainstream software production, we believe that progress must be made in the following six areas:

- Model-Driven Engineering (MDE): this is the idea that, in the future, software systems will not be constructed by writing lines of program code, but at a higher level of abstraction, using appropriate design models;
- Program-It-Yourself (PIY): this is the idea that, in the future, business end-users should be able to specify the kinds of systems they need, using domain-specific languages tailored to express their requirements directly;
- Scrap-Heap Challenge (SHC): this is the idea that, in the future, software will be cannibalised from existing systems, but not at the code-level, rather at the model-level, from which new systems will automatically be generated;
- Self-Verification and Test (SVT): this is the idea that, in the future, models will be correct-by-construction, generated code will be automatically tested against the models, and be self-monitoring for runtime correctness;
- Hit the Moving Target (HMT): this is the idea that software tools and methods used to assemble systems will be agile and flexible enough to achieve all the above, while the requirements are constantly changing;
- Business as Usual (BAU): this is the idea that, no matter what changes are made to the system, business data will be kept safe and there will be no interruptions to normal service.

Now it is clear that a start has been made in some of these areas (hence the focus of this conference), but the point being made here is that unless the whole vision for remodelling software systems is accomplished, then we will never break the back of the software engineering problem. As in all academic disciplines, there is the danger that individual research efforts will carve out small, specialist, fashionable niches that tackle one small area in isolation, without considering the whole picture. In section 3, we argue that this is already happening in MDE, leading to blind spots in the way we approach modelling and model transformations.

3 Taking Control of our Direction

The field of Model-Driven Engineering (MDE) is currently dominated by the Object Management Group's drive towards a specific Model-Driven Architecture (MDA) [10], which explicitly integrates other OMG standards such as the Unified Modeling Language (UML) for its notation [11], the Meta-Object Facility (MOF) and Object Constraint Language (OCL) for bootstrapping its syntax and semantics [12, 13] and the rule-based graph-matching Query-Value-Transformation (QVT) strategy for its model transformations [14]. This has tended to set the agenda for MDE research, much of which either explicitly seeks to encode UML, OCL and QVT, or at least pays lip service to these standards.

3.1 The Unified Modelling Language

How good is UML [11] as the basis for a general-purpose modelling language for MDE? At the moment, we think it is still not ready; and it is in danger of going in the wrong direction. The concerted effort that brought about the substantial revision of UML 2 was intended to "prepare for MDA" in a number of ways, adding OCL at the core for capturing specifications (a good thing), and included major redevelopment of the dynamic notations for describing execution. This finally made "executable diagrams" possible; however, we argue, at the expense of abstraction.

For example, the *sequence diagram* was enriched with labelled frames known as *fragments* that modelled pieces of block-structure, such as the *alt* frame for selection, or the *loop* frame for iteration, or the *ref* frame for referring to another sequence diagram, supporting procedural decomposition. As a result, it became entirely possible to program in sequence diagrams. Yet it is hard to think of a context (except to demonstrate the previous sentence) where this would be a *desirable* way to develop programs. It would be faster to write the code. Furthermore, sequence diagrams are fragile, in that they depend on stable patterns of object collaboration, which are usually the last things to settle during evolutionary system development. So sequence diagrams have little value for abstract design.

Similarly, the *activity diagram* was substantially redeveloped to model procedural systems, independently of the object-oriented perspective (a good thing). It now presupposes a language of primitive *actions* and compound *activities* constructed from these. The activity diagram is fully compositional and could have been mapped onto any process calculus capable of capturing the subtleties of its asynchronous termination, such as CSP [15]. However, the extension to include *object flows* missed a golden opportunity to provide UML with a proper dataflow perspective. The UML notation badly needs an abstract way of describing *what* information flows exist in a system, independently of *when* these flows occur. Instead, object tokens are treated as another kind of Petri token, such that procedural design is complicated by the need to construct the right kind of push-semantics to trigger actions in the desired order [16]. Useful concepts such as data stores are not what they seem in UML 2 (the *datastore* node is in fact merely a kind of inexhaustible buffer). The *activity diagram* is therefore not useful for abstract procedural design; and its alliance with Petri Nets has ruled out other, simpler procedural semantics.

So UML has lost its way, sacrificing clarity for ever-increasing complexity, in the hope of providing (at least one set of) executable semantics. Similar issues exist with *state machines* and the *use case model* [17]. We argue that UML should be going in the opposite direction, aiming for simpler, abstract models that are closer to what stakeholders will understand (for the sake of PIY and SHC). Instead of investing heavily in explaining current UML, we should be aiming to simplify it, and find other ways of generating the desired execution semantics by folding together different modelling perspectives during the transformation process.

3.2 Model Transformation Approaches

How good is current model transformation research? We think this area is still very young and maturing. A number of comprehensive surveys [18, 19] have described the different dimensions of the field. The first concerns whether transformations go forward from models to code, backward from code to models, or are bi-directional (round-trip). The second concerns whether transformations are expressed in algorithms imperatively, or as declarative rules, or as formally proven specifications. The third concerns the number of metamodels used to define the source and target domains and possibly also the transformation process itself.

Current examples show that we can map trees onto graphs, or UML class diagrams onto normalised logical data models. The QVT agenda [14] places a high value on declarative graph transformations, so emphasis is placed on extending imperative approaches with OCL-style comprehensions [20], designing new OCL-flavoured rule-based languages [21] or mapping pure OCL into the imperative transformation steps required to achieve the postcondition [22]. The latter also manages to serialise the order in which rules are applicable, by careful choice of preconditions. Rule ordering is known to be troublesome, from 1970s research in expert systems, so we should be wary of non-determinism in large rule sets. However, we believe that this early fascination with declarative rules is a distraction, when there are still large gaps in our basic understanding about what model transformations are required to generate systems automatically from partial models giving orthogonal views of a system.

As an example of the kind of problem we need to solve, how do you map from a data model, a business process flowchart and some entity life histories onto a set of idiomatic classes and methods that implement the model? By this, we do not mean to build an interpreter, as for BPEL [23], that simulates the models of the flowchart or state machines, but a proper idiomatic translation into Java, Eiffel, C++ or C#. How do you automatically choose the right architecture and assign activities and actions to the appropriate kinds of class and method? How do you generate the missing implementation details for standard business information systems, which surely must follow a small number of generic patterns? How do you ensure that the same abstract models execute with the same semantics in different target languages?

We also believe that model transformation could learn from past work in pattern recognition [24]. Instead of seeking to jump from models to code in one, or two steps (influenced by the OMG's PIM and PSM architectural levels [10]), we should perhaps be thinking of more gradual transformations over several intermediate levels, at which natural design constraints may be exploited to fill in the gaps with boilerplate details.

We foresee the possibility of entirely mechanised intermediate models, which provide the desired execution semantics, but which are derived indirectly from the different partial, abstract and orthogonal views of a system, using a yet-to-be-invented model-folding technique inspired by code-folding techniques from aspect-orientation.

3.3 Satisfying the Business Stakeholders

How good are we at satisfying our customers? There are so many examples of technically competent systems that nonetheless failed to satisfy the end-users' needs [3, 4], to warn us that we software engineers are too technology-obsessed and do not pay proper attention to the stakeholders. Partly, this is just poor communication, but more often due to failure to understand the ergonomics of the business. For example, the antiquated NHS patient record card system worked as well as it did, because the bundle of documents moved with the patient, and doctors and nurses merely consulted this and added their notes as they went. Replacing this by a computer terminal to enter web-form data simply will not work, since doctors cannot afford to keep task-switching while on their clinical rounds.

Socio-technical systems require properly trained customer-facing engineers. This is a perennial problem that is not adequately addressed on computer science courses. The social aspects can only be addressed by improved training. On the technical side, we need much simpler, customer-facing models (see PIY), capable of being faithfully transformed into code (unlike the current position where the UML is either too complicated, or so simple that it is divorced from the real implementation [4]). These cleaner models must have single-viewpoint perspectives [17]; and clearly require much smarter tools capable of folding them together.

Another issue we must take seriously is the rapid pace of change. It will not be acceptable in future to "fix the requirements" at some point and expect these not to change. System modifications will inevitably come and must be expressed at the model-level (see SHC). Partly this is so that the stakeholders can clearly understand and communicate the desired changes; mostly it is necessary because the engineers will not have time to make pervasive changes at the code-level, which will be too dense to comprehend. Again, this means that round-trip engineering methods must eventually be surpassed by confident forward- engineering from models to systems.

Finally, we must take seriously the need to keep the evolving systems live, in continuous operation (see BAU). The notion of the "hot fix" works successfully in some domains (such as web-services), but not in most enterprise-level systems. Data preservation and migration is a serious issue. To some extent, this is starting to be addressed through self-describing data (XML), but the challenge here is to map data faithfully across different ontologies and find data access mechanisms that are still as efficient as relational databases. How can we unpack, describe and repack data in optimal searchable formats? We might take inspiration from CommonLisp's mechanism for updating obsolete objects on-the-fly to fit the revised definition of their class, using metaprogramming to deal with deleted and inserted fields [25].

4 A Call To Action

The above arguments have attempted to demonstrate how far we are from achieving the goals of software engineering. In our vision for software systems remodelling (see 2.2), we identified six areas in which substantial progress must be made. So, what should we do next in the MDE community to help achieve this?

We think that, in order even to understand conceptually the more difficult model transformations that require folding together abstract models from several orthogonal perspectives, we must start with a simpler, semantically transparent set of models. We therefore issue a proposal for μML, the micro-modelling language, which will start from simple, single-perspective [17] models having the following properties:

- compositional task-centred models of requirements, where task composition has simple procedural semantics (sequence, selection, iteration, concurrency);
- control flow models that are derived from the task-centred requirements models, and which have an exact dual relationship with state machine models;
- dataflow models that are independent of control flow, describing *what* information flows exist in the system, rather than *when* such flows happen;
- state machine models with simple reactive Mealy semantics that are the exact dual of control flow models, used to model life histories and user interfaces;
- data models designed to capture the data dependency view for normalisation, not conflated with behavioural dependency brought by object messaging.

We think that model transformation research could then focus on problems involving the folding together of pairs, and later triples, of these models as the folding problem became better understood. This would give rise to the kinds of mechanised intermediate models (as yet largely unexplored and unknown), which capture the space of possible execution semantics. We expect that this will make it possible for standard frameworks to emerge, in which robust boilerplate implementation strategies will finally become possible. These will generate close to optimal code, in styles that follow the natural idioms of each target programming language and platform.

Once we have established conceptually how to achieve the complete forward-transformation process, we can focus on proving formally that the transformations are correct and complete. Once we are certain that the complete, automatic model-to-system translation is correct and robust, we will finally be able to focus completely on the needs of the stakeholders, supporting their views of the information systems that underpin their businesses.

References

1. Johnson, B. and Hencke, D.: Whitehall Wastes 2bn on Abandoned Computer Projects. The Guardian, Friday 4 January, London (2008)
2. Savage, M.: Labour's Computer Blunders cost 26bn. The Independent, Tuesday 19 January, London (2010)
3. Charette, R.N.: Why software fails. IEEE Spectrum, New York, September (2005)
4. McManus, J. and Wood-Harper, T.: A Study in Project Failure. IT Management, British Computer Society, London, (2008), http://www.bcs.org/content/ConWebDoc/19584

5. Hoar, C.A.R.: The Origins of the Grand Challenges. In: Kavanagh, J. and Hall, W. (eds) Grand Challenges in Computing Research 2008, Final Report, UKCRC/BCS/IET (2009)
6. UK Computing Research Committee. Grand Challenges in Computing Research, UKCRC/BCS/IET (2011), http://www.ukcrc.org.uk/grand-challenge/
7. Maude, Rt. Hon. F.: Government ICT Strategy, March 2011, Cabinet Office (2011)
8. Stephens, M. and Rosenberg, D.: Extreme Programming Refactored: the Case Against XP. Apress (2003)
9. Rowe, B.R., Wood, D. W., Link, A. N. and Simoni, D. A.: Economic Impact Assessment of NIST's *Text REtrieval Conference* (TREC) Program, Final Report, NIST/RTI, July (2010)
10. Object Management Group: MDA Guide, Version 1.0.1, Miller, J., Mukerji, J. (eds.), 12 June (2003)
11. Object Management Group: Unified Modeling Language (OMG UML) Superstructure, Version 2.3, 5 May (2010)
12. Object Management Group: Meta Object Facility (MOF) Core Specification, Version 2.0, 1 January (2001)
13. Object Management Group: Object Constraint Language, Version 2.0, 1 May (2006)
14. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.0, 3 April, (2008)
15. Hoare, C. A. R.: Communicating Sequential Processes, Prentice Hall (1985)
16. Bock, C.: UML 2 Activity Diagrams, Part 4: Object Nodes. Journal of Object Technology, 3 (1), 27--41 (2003)
17. Simons, A. J. H. and Graham, I.: 30 Things That Go Wrong in Object Modelling with UML 1.3. Chap. 17 in Kilov, H., Rumpe, B., Simmonds, I. (eds), Behavioural Specifications of Businesses and Systems, Kluwer, 237--257 (1999)
18. Mens, T. and van Gorp, P.: A Taxonomy of Model Transformations. ENTCS, vol. 152, Elsevier, 125--142 (2006)
19. Biehl, M.: Literature Study on Model Transformations. Technical Report, Embedded Control Systems, KTH, Stockholm (2010)
20. Kermeta: Triskell Metamodelling Kernel, http://www.kermeta.org/
21. ATL: A Model Transformation Technology, http://www.eclipse.org/atl/
22. Lano, K. and Rahimi, S. K.: Specification and Verification of Model Transformations using UML-RSDS. In: Mery D., Merz, S. (eds). IFM 2010. LNCS, vol. 6396, pp. 199--214. Springer, Heidelberg (2010)
23. OASIS: WS-BPEL 2.0, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
24. Marr, D.: Vision: A Computational Investigation into the Human Representation and Processing of Visual Information. Freeman, New York (1982)
25. Kiczales, G., des Rivières, J. and Bobrow, D. G.: The Art of the Metaobject Protocol. MIT Press (1991)