Science of Computer Programming 103 (2015) 88–113

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Formal model-driven engineering of critical information systems

Jim Davies, David Milward, Chen-Wei Wang, James Welch *

Department of Computer Science, University of Oxford, Oxford OX1 3QD, UK

ARTICLE INFO

Article history:
Received 17 June 2013
Received in revised form 11 November 2014
Accepted 12 November 2014
Available online 27 November 2014

Keywords:
Model-driven engineering
Formal methods
Critical systems
Information systems
Data migration

ABSTRACT

Model-driven engineering is the generation of software artefacts from abstract models. This is achieved through transformations that encode domain knowledge and implementation strategies. The same transformations can be used to produce quite different systems, or to produce successive versions of the same system. A model-driven approach can thus reduce the cost of development. It can also reduce the cost of verification: if the transformations are shown or assumed to be correct, each new system or version can be verified in terms of its model, rather than its implementation. This paper introduces an approach to model-driven engineering that is particularly suited to the development of critical information systems. The language of the models, and the language of the transformations, are amenable to formal analysis. The transformation strategy, and the associated development methodology, are designed to preserve systems integrity and availability.

1. Introduction

1.1. Failures of critical systems

Our society is increasingly dependent upon the behaviour of complex software systems. Errors in the design and implementation of these systems can have significant consequences. In August 2012, a ‘fairly major bug’ in the trading software used by Knight Capital Group lost that firm $461m in 45 minutes [1]. A software glitch in the anti-lock braking system caused Toyota to recall more than 400,000 vehicles in 2010 [2]. The total cost to the company of this and other software-related recalls in the same period was estimated at $3bn. In October 2008, 103 people were injured, 12 of them seriously, when a Qantas airliner dived repeatedly as the fly-by-wire software responded inappropriately to data from inertial reference sensors [3].

In critical systems development, processes are put in place to detect errors and to mitigate their effects. Incidents such as those listed above have many contributing causes, many of which will be identified as failures of process. For example, only one of the eight causes given for the loss of the $125m Mars Climate Orbiter satellite [4] was directly related to development: “verification and validation process did not adequately address ground software”; the others were failures of communication and procedure. There is nevertheless considerable advantage to be gained from the adoption of automatic tools and techniques that promote correctness in development; if these can be used to eliminate certain kinds of error,

* Corresponding author.
E-mail addresses: Jim.Davies@cs.ox.ac.uk (J. Davies), David.Milward@cs.ox.ac.uk (D. Milward), jackie@cse.yorku.ca (C.-W. Wang), James.Welch@cs.ox.ac.uk (J. Welch).

http://dx.doi.org/10.1016/j.scico.2014.11.004

then the dependency upon other processes is reduced; expensive, manual effort can be focused upon surrounding issues of management and validation.

1.2. Critical information systems

Information systems typically contain large amounts of valuable data subject to complex constraints. The value of the data will usually exceed that of the system: the consequences of data loss, data corruption, or inappropriate access may be unthinkable. This is not simply a matter of deletion. Consider, for example, a clinical system that holds information about doctors and patients. Such a system might hold, as part of a staff record, a list of patients allocated to a particular doctor. The allocation of a doctor may appear also as part of a patient record. If the system reaches a state in which patient A appears on the staff record for doctor B, but doctor B is not listed on the patient record for patient A, then information has been lost. If that information were required for a subsequent decision, then the consequences could be unfortunate.

The constraints upon the data in the system may be representation or type invariants: if these fail to hold, then the behaviour of the system may be unpredictable. Alternatively, they may be semantic constraints or business rules. The symmetric relationship between allocation data on patient and doctor records is a simple example of a semantic constraint: if it fails to hold, then the meaning of the data within the system is unclear. A constraint that no patient is allocated to more than one doctor might be seen as a business rule: if the allocations are properly, symmetrically recorded, then the meaning may be clear, but the rule has been broken. This may mean that some function on the system fails to work as expected. These categories are not exclusive: depending upon the implementation architecture, some business rules may correspond also to representation invariants.

1.3. Formal, model-driven engineering

We might hope to establish, by way of formal specification and proof, that each operation on a system could be guaranteed to satisfy the data constraints. We might proceed by: writing a formal specification *Op* for each operation; writing a formal specification *C* for each constraint; and proving that *Op* preserves the conjunction of all *C*s. However, this approach has three, significant shortcomings: the specification of an operation may not be accurately reflected in its implementation; the system requirements, and hence the data constraints, may change, requiring that any existing proof be revisited; the proof may be difficult and expensive. It is also quite likely, on any particular iteration, that *Op* does not satisfy *C*, and one or both specifications will need to be revised.

A scalable model-driven approach to development, in which software artefacts are generated automatically from precise, abstract models, offers a potential solution. If the implementation of each operation is generated automatically from its specification, and if the generation process is correct, then we can be sure that the specifications are correctly implemented. If the data constraints can be translated and incorporated, automatically, as part of the specifications, then we can be sure that these constraints will be satisfied in operation.

The initial expression of the constraints, and the initial specification of each operation, would still need to be validated. Furthermore, some formal proof may be required to determine whether the results of the translation process—extended versions of the operation specifications that are guaranteed to satisfy the data constraints—correspond to expectations. However, this validation and proof can be conducted at a higher level of abstraction, with many details delegated to a once-only proof of the generation and translation processes. The cost of developing critical information systems, where data integrity can be guaranteed and functionality is predictable, is thus greatly reduced.

1.4. This paper

In this paper, we introduce a formal language, *Booster*, for the specification of data constraints and operations upon information systems. This language is inspired by earlier formal techniques, and set in the context of object-oriented design. We use the Z notation [5] to show how abstract models of information systems can be translated into more concrete designs, and then compiled into complete, working implementations, and how the processes of translation and compilation can be formally verified. We present a simple methodology for the iterative development of critical information systems. We discuss the application of the methodology in the context of critical systems development, exemplified by the development of a system for patient monitoring and self-management in long-term conditions.

This paper builds upon a contribution to the Formal Techniques for Safety Critical Systems workshop (FTSCS 2012) [6]: a methodology for establishing the correctness of the compilation process, from concrete design to working implementation, in the context of critical systems. That paper contains additional details of the generation of SQL code; this paper presents a substantially updated version of the methodology, extended to address transformation from abstract design to implementation. That paper contains a worked example based upon a hotel booking system; this paper contains an extract from the design of an actual critical information system developed using Booster: fully implemented, and relied upon by thousands of users.

This paper complements a contribution on *Success Stories in Model-Driven Engineering* [7]: a report upon the application of the approach to the development of three different information systems. In that paper, we outline the original version of the Booster language and methodology and discuss lessons learned in its application to three case studies. The version