Model-Driven Automation for Simulation-Based Functional Verification

ÉAMONN LINEHAN, EAMONN O’TOOLE and SIOBHÁN CLARKE, Trinity College Dublin

Developing testbenches for dynamic functional verification of hardware designs is a software-intensive process that lies on the critical path of electronic system design. The increasing capabilities of electronic components is contributing to the construction of complex verification environments that are increasingly difficult to understand, maintain, extend, and reuse across projects. Model-driven software engineering addresses issues of complexity, productivity, and code quality through the use of high-level system models and subsequent automatic transformations. Reasoning about verification testbench decomposition becomes simpler at higher levels of abstraction. In particular, the aspect-oriented paradigm, when applied at the model level, can minimize the overlap in functionality between modules, improving maintainability and reusability. This article presents an aspect-oriented model-driven engineering process and toolset for the development of hardware verification testbenches. We illustrate how this process and toolset supports modularized design and automatic transformation to verification environment-specific models and source code through an industry case study.

Categories and Subject Descriptors: D.2.2 [Software Engineering]: Design Tools and Techniques; B.6.3 [Logic Design]: Reliability and testing; Design aids

General Terms: Design, Languages, Verification

Additional Key Words and Phrases: Model-based software engineering, theme/UML, code generation, aspect-oriented, e hardware verification language, function verification

ACM Reference Format:

1. INTRODUCTION

Advances in design automation and semiconductor manufacturing have resulted in the potential to build increasingly complex electronic systems that consequently are difficult to develop and test. The development process is constrained by a need to test early because of the increased cost of correcting faults after hardware has gone into production [Murphy 2010].

Verification can be performed using either formal verification tools or through simulation with functional verification testbenches. The most widely used approach is dynamic and simulation-based, where random or directed stimulus is applied to the design by a testbench and a set of assertions are used to check the resulting behavior for conformance with the specification.

This work was supported in part by Science Foundation Ireland grant 03/CE2/I303-1 to Lero—The Irish Software Engineering Research Centre (www.lero.ie).

Authors’ addresses: É. Linehan (corresponding author), E. O’Toole, and S. Clarke, Lero—The Irish Software Engineering Research Centre, Trinity College Dublin, Dublin, Ireland; email: eamonn.linehan@scss.tcd.ie.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2012 ACM 1084-4309/2012/06-ART31 $15.00
DOI 10.1145/2209291.2209304 http://doi.acm.org/10.1145/2209291.2209304

Figure 1 illustrates a typical hardware development process with verification being performed on the design before the fabrication phase begins [Hamid 2010; Wang et al. 2009]. Simulation-based functional verification consists of developing a testbench and set of testcases. Typically, testbenches perform the following tasks:

— instantiate the Design Under Test (DUT),
— stimulate the DUT by injecting stimuli,
— collect output results and optionally compare them to expected results.

Functional verification testbenches are typically written in domain-specific verification languages such as VHDL, Verilog, e, or OpenVera [Bunker et al. 2004; Engel and Spinczyk 2008], but may also include external data files or C routines [Yogesh et al. 2009; Bergeron 2003]. The development of these testbenches is a software-intensive process that has been reported by Bergeron to consume 70% of the total development time [Bergeron 2003], while in 2007, Li et al. asserted that up to 80% of design costs in many circuit design projects are due to verification [Li et al. 2007].

The large proportion of design time consumed puts verification on the critical path of a design process that is increasing in complexity and cost [Grose 2010]. In addition, the majority of design flaws are functional or logic-related (78% and increasing) and current design processes and tools are achieving a first silicon success of only 28% with a downward trend [Foster 2010].

As systems become more complex, industry demands that verification engineers complete exponentially more complex verification projects in shorter time periods. In electronic systems design, this problem is commonly referred to as the “productivity gap”. As complexity grows, the productivity gap is the difference between hardware capacity and engineering output [Cadence Design Systems, Inc. 2010]. With such a large proportion of design time spent on verification, reducing the productivity gap requires verification engineers to shorten the development time without compromising quality. As functional verification is largely a software engineering task we can turn to the accepted software engineering approaches of increasing productivity by: (1) increasing verification component reuse; and (2) working at higher levels of abstraction.

There have been a number of recent efforts to address the problem of reusability in functional hardware verification [Verisity Design Inc. 2005; OVM World 2007].
However, reusing verification IP is challenging, not least because of incompatible methodologies from single vendors targeting single verification environments, poorly documented verification components, and a need to customize verification components provided by third parties. In addition, many verification components have evolved over years, developed by different teams at different sites, with many layers of new functionality added over a period of time, with knowledge lost along the way through incomplete documentation [Galpin et al. 2009].

The second approach to increasing productivity is to work at higher levels of abstraction where reusing components is made simpler by hiding implementation detail. This approach has been used to enhance the productivity of electronic systems designers by allowing them describe an object (for example, a logic gate made of transistors) using a model where some low-level details are ignored. By applying this approach, digital electronic design went from drawing layouts, to transistor schematics and logic gate netlists, to today’s Register Transfer Level (RTL) descriptions [Zurawski 2004]. Similarly, verification engineers have used hardware environments that have evolved from low-level C libraries to aspect- and object-oriented domain-specific languages for the verification of RTL designs. These environments have built-in support for functional coverage measurement, constrained random generation, and other verification-specific functionality. However, tool support remains limited, and the volume and complexity of code is increasing.

Model-Driven Engineering (MDE) is a design approach where systems are specified as models1. Depending on the level of abstraction of the model, code can be generated ranging from system skeletons to complete, deployable products. MDE raises the level of abstraction at which developers work, promising improved quality and increased productivity through automation [Yusuf et al. 2006]. Companies such as Thales perceive the benefits as (among others) improved productivity, ease of reuse, and independence from technology changes [Koudri et al. 2009]. In the automotive electronics domain specifically, the architecture is evolving along the model-driven route, with companies like Telelogic producing an Autosar model-driven development environment based on UML and SysML, and TNI Software producing Autosar Eclipse/EMF-based tools for model transformation centered on AADL (Architecture and Analysis Definition Language) [Bezivin et al. 2010]. However, to date, the design of verification testbenches has benefited little from the use of model-driven engineering [Shokry and Hinchey 2009].

This article presents an MDE process and toolset that supports modularized development of hardware verification testbenches. Our model-driven development process and toolset includes: a metamodel that facilitates the design of verification testbenches at a higher level of abstraction using a well-accepted set of verification concepts; a model transformation tool that decreases the level of abstraction to produce an aspect-oriented model containing enough details for automatic generation of an executable testbench; a metamodel facilitating the definition of models capturing the full set of concepts from the e hardware verification language; and a code synthesis tool that performs template-based e code generation. In addition, tools have been developed for the eclipse development environment that parse e code to calculate software metrics or generate UML models.

The rest of this article is structured as follows. Section 2 summarizes related work. Section 3 provides a background to aspect-oriented modularization and model-driven engineering. Section 4 describes a formative study that analyses how the challenges identified are manifesting themselves as difficulties in maintaining existing hardware

1The terms model-driven development, model-driven engineering, and model-driven architecture refer roughly to the same software engineering approach and are used interchangeably in the literature, though model-driven architecture generally refers to OMG’s standards-based approach.
verification testbench code. Section 5 describes our model-driven approach to the development of testbench environments for functional verification. Section 6 presents the results of the application of the approach to four case studies. Section 7 concludes the article.

2. RELATED WORK

Modeling Embedded Systems. There have been several projects that applied UML to the modeling of embedded systems using UML profiles, which provide a means of extending UML for a target application domain. Embedded systems have been modeled at the system level through UML profiles that extended the UML semantics to support the specification of system-on-chip designs [Ben Atitallah et al. 2007; Pimentel 2008; Robert and Perrier 2010]. Riccobene et al. have presented a development environment for the hardware/software codesign of embedded systems based on UML and SystemC [Riccobene et al. 2009]. The approach supports graphical specification of SystemC designs in UML and generation of SystemC code from the UML models. Other projects extended the standard UML with profiles supporting embedded systems concepts [Mura et al. 2008; Pampagnin et al. 2008].

Modeling Hardware Specification and Verification. Other work targeted hardware definition and verification languages. Thompson and Williamson used standard UML class diagrams to generate code stubs for the verification language, Vera, using UML to C++ code synthesis [Thompson and Williamson 2002]. The code skeletons are modified by hand to remove C++-specific artifacts and have their behavior inserted. McUmber and Cheng also make use of UML class and state diagrams to specify both structure and behavior [McUmber and Cheng 1999]. In their case, VHDL specifications are generated by applying a set of rules for mapping from UML to VHDL. The UML to SystemVerilog synthesis proposed by Li et al. extends UML with a profile supporting the modeling of real-time systems and informally specifying verification assertions at the model level [Li et al. 2007]. UML state diagrams are then transformed using an intermediate XMI representation to SystemVerilog code. Other tools have applied OMG's MDA approach to hardware description and verification languages [Roussel et al. 2005; Coyle and Thornton 2005; Bunse et al. 2007; Porter et al. 2009; Ben Atitallah et al. 2007].

These early examples of modeling hardware definition and verification languages fail to hide implementation details from the model level. They do, however, increase the level of automation and facilitate the specification and transformation of models at multiple levels of detail. The value in providing models of hardware verification environments was demonstrated by Gluska and Libis in the development of their pre-RTL model for verification [Gluska and Libis 2009]. They identified verification as being on the critical path of hardware development and demonstrated how abstract modeling of designs can shorten the verification process and enhance the effective use of coverage and formal verification techniques. However, the modeling approach taken was textual and did not consider reusability or modularity.

Modeling Multiprocessor Systems. Oliveira et al. describe a model-driven approach for MPSoC design space exploration that focuses on addressing the issue of reasoning about complex multiprocessor systems [Oliveira et al. 2007]. The tool supports a UML-based approach to design space exploration and an automatic multiobjective design space estimation mechanism. Support for model transformation is provided following the exploration phase of the development cycle, although it is unclear whether automated support for code generation is provided.

Modeling Verification Testbenches. The TestBencher Pro graphical code generator by SynaptiCAD Inc., inspired by model-driven engineering’s platform-independent models, provides a means to model verification testbenches independent of the verification language in use [SynaptiCAD 2010]. However, TestBencher Pro’s timing diagram can...
only model a subset of the functionality required of a verification testbench. Similarly, Telelogic Rhapsody Developer in C2 is a model-driven engineering tool that generates C code based on UML 2.0 behavioral and structural diagrams. UML models in Rhapsody Developer conform to the FunctionalC UML Profile [Douglass 2008].

Aspect-Oriented Modeling. These examples of model-driven engineering approaches reduce design complexity by raising the level of abstraction at which engineers work. However, simply raising the level of abstraction does not necessarily result in improved comprehensibility and reusability. These properties are often determined by how well concerns are modularized in the design. Aspect-oriented design has been proposed as a better modularization for cross cutting concerns in embedded systems [Driver et al. 2010; Loukil et al. 2010].

There has been some work in aspect-oriented and model-driven engineering of embedded systems [Afonso et al. 2008; Wehrmeister et al. 2007; Bergeron 2003; Gérard et al. 2005; Gomes et al. 2009; Diaz-Herrera et al. 2002]. However, these approaches cannot be directly applied to functional verification as hardware verification languages incorporate constructs that do not appear in general-purpose high-level languages (like C++ or Java).

In general, related work provides modeling constructs for related domains such as embedded systems, hardware specification, or multiprocessor systems, but not hardware verification testbenches. They also provide for aspect separation at the model level, but again, do not support hardware verification. This article provides both with an approach to managing the increasing complexity of hardware verification testbenches through the application of an aspect-oriented modularization at the model level. This approach is supported by a model-driven engineering process, a set of modeling languages, model transformations, and tools.

3. BACKGROUND

The modeling approach described in this article draws on aspect-oriented and MDE techniques. Here, we present a brief introduction to these concepts.

3.1. Aspect-Oriented Modularization

Object-Oriented Programming (OOP) offers a “separation of concerns” that allows designers to break down a program into distinct parts. Each of these parts, be they classes, packages, components, etc., are designed to encapsulate all the code related to a single concern. However, object-oriented decomposition results in developers having to work on many concerns at once as secondary concerns crosscut the primary decomposition, resulting in modules that overlap in functionality. Allan et al. presented a simple example of a single task from a class used to manage a DMA controller [Allan et al. 2004]. In his example, code for tracing the execution of the program, handling errors, checking input, and accessing shared resources was scattered throughout the code dealing with the task’s intended function. The code dealing directly with the task’s intended function is referred to as the dominant concern and the remaining code belongs to secondary crosscutting concerns.

Aspect-Oriented Programming (AOP) offers a new construct, an aspect, that can be used to encapsulate these crosscutting concerns in a way that minimizes the overlap in functionality between modules [Kiczales et al. 1997]. Dominant concerns are coded using OOP as before and aspects are used to code crosscutting concerns and specify their integration with the dominant concerns. Studies have shown that AOP improves the degree to which crosscutting concerns are separated in software, improving
maintainability and developer productivity [Greenwood et al. 2007; Walker et al. 1999; Bartsch and Harrison 2008].

The AOP features of the $e$ language give it the power to significantly simplify and accelerate the development of reusable, automated verification environments [Robinson 2008]. However, aspect-based techniques in $e$ are rarely considered at design time as a way of modularizing code. Instead, aspect-oriented features in $e$ are often used to add new features to existing code without having to intrusively modify the code base [Iman and Joshi 2004].

3.2. Model-Driven Engineering

Model-driven engineering is an approach to software development that focuses on the production of high-level models that are used as the basis for automating system implementation. The fundamental notions behind MDE are to raise the level of abstraction of software specifications away from underlying implementation technologies and to automate the transition from design specifications to corresponding implementations [Selic 2008].

Standards for model-based design emerged with the publication of the Object Management Group’s Model-Driven Architecture (MDA) [Miller and Mukerji 2003]. Although MDA originally targeted large-scale enterprise applications, its underlying principles have been widely adopted and applied to the development of applications in a diverse range of domains, including hardware design and verification. The MDA approach can be characterized by Platform-Independent Models (PIMs), derived from requirements that are transformed into one or more Platform-Specific Models (PSMs) for the actual implementation. The advantage of the MDA approach is that incremental, iterative development is facilitated by the direct transformation from model to code. Increased automation in the production of system code reduces the potential for the introduction of human errors. Models are captured using modeling languages such as UML, which includes a range of structural and behavioral modeling constructs.

In the MDA standard, provision has been made for extending the capabilities of the modeling concepts through the use of profiles. Profiles are extensions to the standard metamodel that allow language designers specify particular constructs of interest in a domain. For example, the MARTE profile provides constructs for specifying real-time and embedded systems [OMG 2008]. Another example is the UML for Testing Profile (UTP), which provides constructs to support testing-related activities [OMG 2011b]. For our work, we extend the standard UML with a profile to capture hardware verification constructs.

Aspect-oriented design complements the MDE approach by facilitating the partitioning of models along aspect boundaries, providing a single view of each concern. Our MDE process incorporates modeling conventions for expressing crosscutting concerns at the model level using Theme/UML [Clarke and Baniassad 2005] into its $e$ UML2 profile and the model-to-model transformation phase that generates these aspect-oriented $e$ models [Linehan and Clarke 2010]. Theme/UML is an aspect-oriented extension to UML that supports fine-grained decomposition and composition of both functional and nonfunctional concerns, including those that are crosscutting. For more details of modeling verification testbenches in the $e$ language, see Linehan and Clarke [2010].

4. FORMATIVE STUDY

In this section, we analyze the structure of a typical production testbench using a set of accepted metrics implemented as a plug-in for the eclipse development environment [The Eclipse Foundation 2010]. This analysis illustrates that current testbench design methodologies [Verisity Design Inc. 2005; Bergeron et al. 2005; OVM World 2007], software engineering lifecycles, verification languages, and tools are resulting in
testbenches that are not optimally designed for easy reuse. While not a direct contribution to our work, this section will provide some context to the evaluation of the testbenches in Section 6, and reinforce the motivation for the model described in Section 5.

The testbenches analyzed in this work are implemented in the $e$ hardware verification language. The $e$ language is a domain-specific programming language that was developed in 1997 by Verisity Design as part of their Specman tool [Iman and Joshi 2004]. $e$ was standardized as IEEE 1647 with a revision published in 2008 [IEEE Computer Society 2008].

As a testbench language, $e$ provides constructs related to stimuli generation, such as specification of input constraints and facilities for data packing and assessing simulation coverage. $e$ also contains constructs that support monitoring and checking the response of the DUT, and constructs to support assessment of the functional coverage of the DUT (as opposed to simply the code coverage).

### 4.1. Metrics

To support our analysis of the industry testbenches, we developed a tool that measures object-oriented and aspect-oriented software metrics for the $e$ language. The tool, illustrated in Figure 2, is a plug-in for the eclipse development environment.

The metrics used are based on those in Hoffman and Eugster [2008], defined for the most widely used aspect-oriented programming language, AspectJ, extended for the $e$ language. Table 1 lists the metrics used, grouped under the headings, size, coupling, and complexity. Size is measured by counting the number of lines of code, modules, and structures in the testbench. Coupling between objects and aspects is measured by analyzing the communication patterns between modules. High coupling is considered detrimental to modular design and prevents reuse [Chidamber and Kemerer 1994]. Complexity is measured by analyzing the size of a testbench (in terms of structure and lines of code) and measuring the degree of connectedness of the testbenches, structures. High complexity is an indicator of low maintainability and understandability, making reuse more costly.

These metrics are used to identify problems related to structure and modularization. Modularization promotes the use of well-defined, independent modules to increase maintainability and comprehensibility. This study provides evidence to support our
Table I. Implemented e Metrics

<table>
<thead>
<tr>
<th>Category</th>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
<td>Lines of Code (LOC)</td>
<td>Size is a measure of effort, cost, maintainability and degree of modularization. LOC is the total number of lines of code (excluding comments) in a program.</td>
</tr>
<tr>
<td></td>
<td>Number of Modules (NOM)</td>
<td>A structural metric that indicates the number of modules in a program, where module is synonymous with component or package.</td>
</tr>
<tr>
<td></td>
<td>Number of Structures (NOS)</td>
<td>A structural metric that counts the number of structures in a program where a structure is an object, interface or aspect.</td>
</tr>
<tr>
<td><strong>Coupling</strong></td>
<td>Coupling Between Modules (CBM)</td>
<td>Coupling or dependency is the degree to which each program module relies on each one of the other modules. CBM is a metric that counts the number of modules that contain fields or operations that may be called from a specified module. High coupling is detrimental to modular design and reduces reusability.</td>
</tr>
<tr>
<td></td>
<td>Coupling on Method Call (CMC)</td>
<td>The number of modules or interfaces declaring methods that are called by a given module. A dependency metric concerned with operations.</td>
</tr>
<tr>
<td><strong>Complexity</strong></td>
<td>Number of Children (NOC)</td>
<td>NOC is a number of immediate subclasses or sub-aspects of a given module.</td>
</tr>
<tr>
<td></td>
<td>Depth of Inheritance Tree (DIT)</td>
<td>DIT is a length of the longest path from a given module to the class/aspect hierarchy root. The deeper a module is in the hierarchy, the more difficult it is to determine its behavior due to the greater number of operations it is likely to inherit.</td>
</tr>
</tbody>
</table>

hypothesis that published methodologies [Bergeron et al. 2005; Iman and Joshi 2004], verification IP [Synopsys, Inc. 2008; OVM World 2007; Verisity Design Inc. 2005], tutorials [Krishna and Maddipati 2010; Tala 2010], and best practice guidelines [Bening and Foster 2002] on testbenches design from industry are resulting in testbenches that are not optimally decomposed. This article presents an approach to address these structural problems by using model-driven engineering to reason about decomposition at higher levels of abstraction and aspect-oriented design to address verification concerns that cannot be cleanly modularized with object-oriented methods alone [Engel and Spinczyk 2008; Allan et al. 2004; Vax 2010; Hollander et al. 2001].

4.2. Benchmark

To place the application of these metrics in context with their application to other software projects, we have selected one of our four industry case studies presented in Section 6, and compared it to other studies. The software projects used for comparison are taken from a case study on AspectJ by Filho et al. [2006], with additional metrics taken from another study featuring the same applications by Hoffman and Eugster [2008].

Figure 3 presents the results of the size metrics on one testbench (Case Study 1 from Section 6.1). Please note that the figure shows a bar chart with more than one y-axis for readability. This testbench has 7290 lines of noncommented source statements, 33 structures (modules, objects, interfaces, aspects), 588 attributes, and 246 operations. These results show that the testbench contains a small number of large structures, suggesting that the design has not been sufficiently decomposed. When compared with other case studies, it can be seen that the testbench contains far less structures, proportional to the number of lines of code.

The complexity metrics (not graphed) show that the testbench makes no use of inheritance. Shallow inheritance trees typically result in a low-complexity design but
an absolute zero depth indicates a completely flat structure that relies on aspect-oriented extension to add behavior to modules. In addition, the testbench contains large operations which are more difficult to understand, debug, and maintain. On the other hand, coupling between modules is in line with other projects, with some scope for improvement.

These findings provide some empirical support for the challenges already reported to us by verification engineers [Galpin et al. 2009]. As reported to us anecdotally, hardware verification testbenches are becoming increasingly complex, difficult to understand, reuse, and maintain. Poor modularization can, in part, be attributed to the reuse methodology [Iman and Joshi 2004] that advocates an OO decomposition at design time and the extensive use of aspect-oriented features in $e$ as a means to extend existing testbenches with new behavior.

5. MODEL-DRIVEN AUTOMATION: THE METHODOLOGY

In this section, we describe our model and illustrate its application on a small sample testbench. Traditional approaches to developing testbenches begin with a design specification that is directly implemented as code by hand. The coding step includes a search for reusable components from existing testbenches and the extension and customization of such components. It is common for coding to begin without producing a detailed design as the implementation is iteratively refined, based on reported coverage, towards a correct implementation.

In contrast, our approach supports correct-by-construction software synthesis through a high-level generic testbench model, from which an aspect-oriented system decomposition is generated. This aspect-oriented model provides better separation of concerns, allowing the designer reason about different functionalities, including those that scatter across a system, separately. This approach complements existing approaches to testbench development by facilitating separation of concerns in design, automated model composition, and source-code generation. The source-code, written in a hardware verification language, remains an important asset with our MDE process, which facilitates better modularization of the code.

This section presents our approach to managing the complexity of hardware verification testbenches through the application of aspect-oriented modularization at the model level, using an industry testbench to illustrate each step.
5.1. Model-Driven Engineering with Aspects

We have defined a development process, illustrated in Figure 4, based on the MDA approach but tailored to address the specific challenges of managing complexity, productivity, and quality in hardware verification testbenches. The process is focused on modularization and contains three distinct phases that are titled based on the activities of the developer during each phase. The first phase involves system modeling via specification of testbench component architecture and testcases (see Section 5.2).

The second phase involves applying a model-to-model transformation to the testbench model to produce a well-modularized HVL-level model containing all the information necessary to generate source code in a specific verification language. This model is defined by extending UML with a new UML2 profile for the e verification language that reuses and extends MARTE [OMG 2008] and Theme/UML [Carton et al. 2009] to enable the aspect-oriented modeling of testbenches in the e language (see Section 5.3).

The final phase of the development process is the synthesis of source code (see Section 5.4). We developed a set of templates using Xpand\(^3\) that encode a UML to e mapping. In addition to code generation, the final phase provides tools that can walk the source tree of generated e code. The metrics tools have already been presented in Section 4.1 and an on-demand round-trip engineering tool [Sendall and Kster 2004] supports the construction of UML models using our e UML2 profile from e source code. For more details of this work, see Linehan and Clarke [2011].

5.2. Modeling

As illustrated in Figure 4, testbench development begins with the modeling of the testbenches’ main concepts using a UML editor. To facilitate the construction of this model, a UML2 profile has been defined that describes the platform-independent constructs and abstractions common to simulation-based functional verification testbenches.

The hardware verification domain has a unique, well-accepted set of concepts, that hardware verification engineers are familiar with from libraries, code generators, and standardization efforts [Dempster and Stuart 2001; Bergeron 2003]. The constructs of interest for our profile were derived from a combination of analyzing the

\(^3\)Xpand is a language for code generation based on EMF models that is part of the eclipse modeling project [The Eclipse Foundation 2010].
structure of testbenches and a literature review. The literature review revealed a slew of methodologies [Bergeron et al. 2005], verification IP [Synopsys, Inc. 2008; OVM World 2007], tutorials [Krishna and Maddipati 2010; Tala 2010], and best practice guidelines [Bening and Foster 2002] on how testbenches can be designed to yield more efficient verification while maintaining quality (measured as the likelihood of first-silicon success). The primary high-level concepts in our profile, sometimes referred to as reusable verification components or verification IP, include the following.

**Agent.** Agents are architectural elements that communicate with all testbench components. Agents contain relationships to a **Driver** to generate traffic and a **Bus** to drive traffic onto the DUT and a **Monitor** for coverage collection and protocol checking.

**Bus.** A Bus is a type of **Port** between a Master and Slave device with a fixed set of **Signals** and a protocol specifying its behavior.

**Checker.** The checker verifies whether data is as expected and returns the comparison state to the **Scoreboard**. The **Checker** is also responsible for unpacking the low-level data (signals) to high-level data (sequences/transactions).

**Clock.** The **Clock** is the reference to the clock generator being used to synchronize the testbench and DUT simulation. Testbench **Events** are tied to this clock.

**Component.** A Component is a top-level structure from which **Drivers**, **Scoreboards**, and other objects in a testbench inherit.

**Constraint.** A **Constraint** is a rule (defined by the hardware specification) that forms part of a **Test**.

**Coverage.** Represents a functional coverage group.

**Driver.** The **Driver** translates the operations produced by the **Generator** into the actual inputs for the DUT. The **Driver** also handles the DUT configuration operations.

**DUT.** The Design Under Test component is a place holder referencing the actual model of this component specified in an HDL.

**Event.** Testbench event.

**Generator.** The Generator produces stimulus that is written to DUT by **Driver** at a high level of abstraction, that is, Sequences.

**Monitor.** The **Monitor** converts the state of the DUT and its outputs to a transaction abstraction level (Sequence) so it can be stored in the **Scoreboard** database to be checked later on.

**Port.** Ports provide the interface used to convey **Transactions**.

**Report.** A message from the testbench to be recorded, usually in the form of a plain text message to be logged but can contain state information.

**ReportHandler.** The reporting classes provide a facility for issuing **Reports** with consistent formatting and configurable side-effects, such as logging to a file or exiting simulation.

**Sequence.** A single transfer comprising a collection of signals. Sequences are often identified by an OPCODE (Signal).

**Signal.** A **Signal** represents a named wire connecting to a DUT input pin.

**Test.** Represents a single testcase. **Tests** are normally related to a requirement in the design specification and consist of a collection of **Constraints**. These Constraints configure the testbench to behave in a particular manner by controlling the Sequences produced by the **Generator**.

**Scoreboard.** The **Scoreboard** stores the input and expected DUT output.

The primary requirement for a UML editor used during the modeling phase is that it supports the use of UML2 profiles and export to Eclipse Modeling Framework\(^4\) (EMF)

\(^4\)http://www.eclipse.org/modeling/emf/.
XML Metadata Interchange (XMI) format. Our current editor of choice is MagicDraw 16.9, which is free under academic license. However, any UML editor that supports integration of profiles and export to XMI can be used.

In the modeling phase, two sets of UML models are produced: a model of the testbench itself; and a model of the test cases which constrain the testbench behavior to test particular features of the hardware design. In both, model elements were extracted from the four case study industry testbenches through manual inspection.

The modeling phase is illustrated by an extract taken from Case Study 1 (Section 6.1)\(^6\) that illustrates some of the modeling constructs. The testbench is a coverage-driven testbench making use of constrained random generation to verify the behavior of a proprietary bus protocol. Figure 5 shows the structural design of the testbench extract.

These model elements include an sri\_drive Driver element which is used to translate inputs for the the bus protocol being tested. The Driver element makes use of an sri\_trans Transfer element, which is used to encapsulate a bus transfer as a data structure. The Transfer element contains attributes, such as bus_width, and methods, such as extract\_data\_bus which are specific to the bus protocol being verified in the testbench. Additional elements include the Agent elements sri\_evc\_agent\_u. The Agent element communicates with all testbench components, using the output from the Driver element to provide input for the bus. It contains the bus that is under test and the ability to write to and read from the bus. Finally, a sri\_log ReportHandler element is also included that writes details of bus transfers to a log file. The majority of the relationships between stereotyped elements are contained within the testbench UML2 Profile [Linehan and Clarke 2011].

5.3. Transformation
The next phase of our MDE process (Figure 4) is a refinement step where transformations are applied to the testbench model to provide a more detailed level of abstraction. In this phase, the UML models developed during the modeling phase are exported from the UML editor as Eclipse UML2 XMI. This model transformation outputs an

\(^6\)Nondisclosure agreements prevent us from including the complete design of the original or reengineered testbenches in all four cases.
aspect-oriented UML model specific to the $e$ verification language, containing details for automatic generation of executable code.

Model transformations may be implemented in different ways. In this article we express transformations using a specialized model transformation language. There are a number of model transformation languages available [Agrawal et al. 2006; Varró et al. 2002], with ATL (ATLAS Transformation Language) currently having the best tool support. ATL is a mature tool that has been deployed in industry and been shown to be capable of solving nontrivial problems [Jouault et al. 2008]. ATL is accompanied by a set of tools (ATL Development Tools (ADT)) built on top of the Eclipse platform and is composed of the ATL transformation engine and of the ATL Integrated Development Environment.

ATL can be used in two different modes: refining or normal mode. Our transformation process uses ATL in normal mode, where an output model is generated by copying and transforming all the information contained in the source model. ATL transformations consist of a collection of transformation definitions. Transformation definitions are grouped into modules that contain a header, import section, and a number of helpers and transformation rules. Helpers and transformation rules are the constructs used to specify the transformation functionality. Our ATL transformation includes helpers that access and apply stereotypes from the $e$ and testbench UML2 profiles.

Figure 6 illustrates the model transformation pattern. In this pattern, a source model, `sample-tb.uml`, is transformed into a target model, `generated-e-tb.uml`, according to a transformation definition, `tb-to-e.atl`, written in the ATL language. The transformation definition is a model conforming to the ATL metamodel. The source model conforms to our testbench UML2 profile. The target model conforms to the $e$ profile, which is realized in `e.profile.uml`. All metamodels conform to the MOF [OMG 2006].

Transformation rules written in the ATL language match elements of the source model and generate from them a number of distinct target model elements. The output of the model-to-model transformation for our illustrative case study is shown in Figure 7. This UML class diagram7 shows a snippet of the generated UML model.
corresponding to the elements included in Figure 5. This excerpt illustrates many of the features of the model-to-model transformation process, in particular, high-level model elements are realized as modules, such as the Driver stereotype in our source model resulting in a `tb::Driver` module in our target model. Some of these modules are crosscutting, indicated by a module dependency relationship with the Extend stereotype applied. To illustrate transformations applied in more detail and the use of aspects in our generated model, we will now discuss the transformations applied to the `Agent` and `ReportHandler` stereotypes in our high-level source model.

A transformation rule maps an `Agent`, sri_evc_agent_u, in our testbench profile (source) into an `e` module, `tb::Application`, containing an `e` Unit, sri_agent_u with copies of the operations and attributes defined in the source class (target). The `Agent` encapsulates a `Bus`, which is defined in the profile as a port connecting a Master and Slave device. These devices are transformed to be two corresponding subclasses of sri_evc_agent_u. The use of inheritance in `e` is illustrated by the Like stereotype applied to the relationship between sri_agent_u and the Units sri_master_agent_u and sri_slave_agent_u. These two Units inherit their behavior but are used to separate the signals for the master and slave side of the bus, respectively. The `tb::Application` module contains the application classes, sri_mif_u, and is the central module of the testbench. `Agent` elements are located here as they communicate with all testbench components. A similar transformation was applied to the sri_trans Transfer element in our testbench profile (source). It results in an `e` Struct sri_transfer_s which retains the attributes from the earlier model.

The use of aspects is illustrated by examining the `tb::Log` module. A transformation rule maps a `ReportHandler` stereotype in our testbench profile (source) into an `e` module, `tb::Log`, containing an `e` Unit with copies of the operations and attributes defined in the source class (target). This results in the `e` Unit sri_log. Additionally, a Driver, driver_u, is also defined with its associated attributes and operations in the `tb::Log` module. The Extend relationship indicates that these attributes and operations are merged with those of the Driver object in the `tb::Driver` module. That is, the `tb::Log`
module is an aspect that, through identically named concepts, is composed with base classes in the \texttt{tb::Driver} module.

This approach ensures that the \texttt{tb::Log} module contains only the structures and behavior necessary to represent the \texttt{ReportHandler} behavior and the \texttt{tb::Driver} module contains only elements related to Driver verification component behavior. This design illustrates how the crosscutting \texttt{ReportHandler} concern is cleanly modularized within the single module as opposed to occurring repeatedly throughout the code. This aspect-oriented approach eases the creation, maintenance, and reuse of testbenches by making it possible to tackle common testbench scenarios that are not easily solved using OOP techniques alone [Allan et al. 2004].

Other crosscutting behavior emerges in the \texttt{tb::Driver} module. A transformation rule is mapped from a \texttt{Driver} stereotype in the source model into an \texttt{e} module containing the \texttt{e Unit} driver. The module extends the \texttt{tb::Application} module with an \texttt{sri mif u} Unit created to run to the Driver, once again modularizing a concern within a single module. This module additionally contains the \texttt{e Unit Entropy Decoder} which allows the Driver to translate inputs for the DUT.

Where Figure 7 focuses on a portion of the generated model, Figure 8 illustrates the full set of unique modules generated in the transformation phase for the illustrative case study testbench. The relationships between each module are included showing how aspect-oriented extensions are used to keep verification components loosely coupled with respect to each other. These relationships dictate how the modules are composed to form a complete testbench and how the base modules are bound to the aspect modules that crosscut them.

5.4. Synthesis

The next step in the MDE process (see Figure 4) is to automatically synthesize source code from this model. In this phase, the UML model (encoded as Eclipse UML2 XMI) is imported into an Xpand Generator eclipse project. Xpand is a language for code generation based on EMF models that is part of the eclipse modeling project [The
Eclipse Foundation 2010]. This project contains the Xpand template, the e metamodel, and a workflow.

Xpand supports the ability to define rich libraries of independent operations and non-invasive metamodel extensions based on Java. This facility is exploited to implement helper functions in Java for generating e constructs and extracting information from the source model. Specifically, Java extensions are used to access the aspect-oriented features of the input model to determine where composition relationships indicate that one module contains features extending another.

The code output for our illustrative case study, by the synthesis phase, is illustrated in Figure 9, with the code generated for the tb::Application from our generated e model in Figure 7. Its constituent elements are displayed. The workflow file specifies a location on disk where generated files should be written. We copied these generated files into an e project, making use of the DVT plug-in for eclipse to provide compilation, code formatting, and navigation.

Applying the complete MDE process resulted in eleven source-code files being generated in the e verification language. These files are organized in a package hierarchy that reflects the structure of the e model with each package containing many files and each file containing a single e module. The next section evaluates this generated code by comparing it with the original testbench developed using more traditional methods by way of the e source-code metrics plug-in presented in Section 4.1.

6. EVALUATION
To assess the applicability of our MDE process and toolset to testbench development we conducted four case studies involving the redesign and reimplementation of sample
industry testbenches [Pilkington et al. 2009]. It was our view that testbenches from industry were the best representation of the state of practice and, therefore, the best base line against which to measure our approach.

### 6.1. Case Study 1

This case study analyses a sample testbench in use at a large multinational automotive semiconductor design and manufacturing company. The testbench is a coverage-driven testbench making use of constrained random generation to verify the behavior of a proprietary bus protocol. It consists of multiple master and slave end points connected to the design under test via a crossbar. The testbench contains 7290 lines of uncommented code and is designed to work with Cadence’s Specman tool and interacts with a design modeled in VHDL. It was contributed to by both internal employees in a number of design offices and various external contractors. Existing components that have been developed externally (referred to as Verification IP) are reused and adapted in the implementation of the testbench.

The case study produced a complete reimplementation of the sample testbench. The generated testbench is, as far as possible, equivalent to the original testbench in terms of verification components and features. By comparing the testbench generated using our aspect-oriented MDE approach to the original testbench from industry it is possible to appraise our approach in terms of improvements to the modularization and decomposition of the testbench.

Figure 10 shows the results of comparing the testbench generated by the case study with the original industry testbench, developed using a conventional process. The two implementations are compared by running each testbench through the $e$ metrics tool (see Section 4.1). It is the goal of our MDE development process, models, and tools to improve the modularization of testbenches as a mechanism for improving the flexibility, comprehensibility, maintainability, and reusability while shortening development time. Through our formative study (Section 4) we have shown poor modularization to be one the causes of the increase in complexity and resulting difficulties in reusing and maintaining testbenches in industry.
The results of this comparison, shown in Figure 10, are grouped into three categories: size, coupling, and complexity\(^8\). The size metrics, \textit{NOS} and \textit{NOM}, indicate that the MDE process has resulted in a testbench with less modules and structs. This is as a result of the original testbench grouping behavior by type rather than concern. Modularization by concern results in fewer modules and structs, which reduces coupling. For example, modules existed for macros, types, and units. This resulted in a range of structures that could be eliminated by modularizing the code based on concerns. Correcting for these differences, the two testbenches would be similar in size.

The \textit{CBM} and \textit{CMC} metrics indicate a better modular design of the refactored testbench due to decreased coupling. Decreasing the coupling has been reported to have the result of increasing maintainability and reusability [Li and Henry 1993; Munnelly et al. 2007]. The two complexity metrics, \textit{NOC} and \textit{DIT}, relate to inheritance. Whereas the original testbench makes no use of inheritance, the generated testbench does make limited use of inheritance. The shallow inheritance tree in the generated testbench makes it easier to infer behavior due to the low number of operations that are inherited. The use of inheritance itself contributes to our goal of improving modularization by compartmentalizing code in a way that makes it easy to reuse.

6.2. Case Study 2

This case study analyses a sample industry testbench that provides driving and monitoring facilities for DDR1- and DDR2-compliant devices. The testbench supports any number of Bus agents and includes protocol checks, sequence libraries, and coverage definitions for BUS transactions. The testbench is designed to work with the Incisive Enterprise Specman tool and contains 7673 lines of uncommented code.

Figure 11 shows the results of comparing the case study’s refactored testbench with the original sample testbench. The size metrics, \textit{NOS} and \textit{NOM}, show both testbenches are of similar size with the refactored testbench having more structs but less modules than the original testbench. The \textit{CBM} and \textit{CMC} metrics indicate that the generated testbench has resulted in a marginally better modular design as the coupling between

\(^8\)See Table I for details of these metrics.
modules has been reduced. Finally the complexity metrics, \(\text{NOC}\) and \(\text{DIT}\), indicate that no improvement has been made in the generated testbench on the complexity of the industry testbench with both testbenches possessing a similarly shallow inheritance tree.

Interestingly, overall the results for the refactored testbench do not show a significant improvement over the original testbench, especially in comparison to Case Study 1, despite the original testbenches in both cases containing a similar number of lines of code. On analysis, a more concern-based approach was adopted in the original testbench in this case study resulting in a proportionately fewer numbers of modules and structs. This limited the scope of improvements that could be made.

### 6.3. Case Study 3

Case Study 3 analyses a sample industry testbench for the I2C (Inter Integrated Circuit) BUS system. The testbench provides full control over I2C symbol generation with both master and slave end points. The testbench also includes protocol checks, sequence libraries, and coverage definitions for I2C BUS transactions. The testbench is designed to work with the Incisive Enterprise Specman tool and contains 4540 lines of uncommented code.

Figure 12 shows the results of comparing the case study’s refactored testbench with the original sample testbench. The size metrics, \(\text{NOS}\) and \(\text{NOM}\), indicate that an improvement has been made in the refactored testbench with the number of modules being reduced from 73 to 47. The remodularization did, however, require the introduction of two additional structs.

The \(\text{CBM}\) and \(\text{CMC}\) metrics similarly indicate that the refactored testbench has a better modular design than the original testbench. The coupling between modules has been reduced significantly and the coupling on the method call also shows some improvement from the original testbench. Finally the complexity metrics, \(\text{NOC}\) and \(\text{DIT}\), indicate that a slight improvement has been made to the complexity of the original testbench as the generated testbench has a slightly shallower inheritance tree and the number of children is also reduced.
In this case study, we observed improvements similar to those achieved in Case Study 1. Despite consisting of only 4540 lines of code, the original testbench contained a larger number of modules and structs than the original testbench in Case Study 2. On analysis, this testbench was not modularized based on concerns and so a large number of modules could be eliminated during refactoring. This in turn leads to improved modularization as the coupling across the testbench was decreased.

6.4. Case Study 4

The final case study analyses a sample industry testbench that provides driving and monitoring facilities for Powerwise Interface (PWI 2.0)-compliant devices. The sample testbench also includes protocol checks, sequence libraries, and coverage definitions for PWI transactions. This testbench is the smallest of the four sample testbenches containing 3006 lines of uncommented code. The testbench is designed to work with the Incisive Enterprise Specman tool.

Figure 13 shows the results of comparing the refactored testbench with the original sample testbench in this case study. The size metrics, NOS and NOM, show that both testbenches are of similar size with the refactored testbench having more structs but less modules than the sample testbench. The CBM and CMC metrics indicate that the refactored testbench has a marginally better modular design than the original testbench as the coupling between modules has been reduced. Finally the complexity metrics, NOC and DIT, indicate that a slight improvement has been made to the complexity of the original testbench as the generated testbench has a slightly shallower inheritance tree.

Similar to Case Study 2 detailed earlier, the refactored testbench here does not demonstrate a significant improvement over the original testbench. On analysis, the original testbench here shared some design characteristics with the the original testbench in Case Study 2, illustrating a much greater concern-based approach than the other two case studies. For example, the original testbench here contained modularized behavior for “monitoring” and “checking”, amongst others. This limited the improvements that could be made by our approach.
6.5. Discussion
The case studies presented before demonstrate that in some cases, our approach can result in a better modularized testbench that will, in turn, be easier to maintain and reuse. This has been achieved without needing to introduce verification engineers to aspect-oriented analysis and design by transforming an object-oriented platform-independent testbench model into an aspect-oriented model based on the naturally crosscutting nature of verification components such as coverage, logging, checking, etc. [Engel and Spinczyk 2008].

The case studies indicate that the biggest improvements are found in testbenches that are designed with little or no concern-based modularization (Case Studies 1 and 3). These testbenches typically contain a proportionately large number of structs and modules with a scattering of behavior throughout the testbench. By refactoring these testbenches using our aspect-oriented approach, it is possible to reduce the number of modules and structs, which in turn helps to decrease coupling and complexity in the testbench.

However, testbenches that are designed with a more concern-based approach, such as those in Case Studies 2 and 4, do not benefit from our approach to the same extent. In these cases traditionally crosscutting behavior will be modularized to a large degree, limiting the benefit of the approach outlined here. Of course, even in these cases, the development process can benefit from the higher level of abstraction that can be supported with models.

As a result of these studies, it would be interesting to explore the boundaries around which it is beneficial for companies to engage in this process. Given that our results show mixed benefits, we advise that the goals for changing an established approach should be carefully considered, and the potential benefits of either improved reuse or modeling with abstractions and using transformations are assessed. From our point of view as researchers, we would like to be in a position to more scientifically define a set of heuristics to aid in this process.

The design of a platform-specific model for the e verification language introduces familiar verification constructs into the MDE process, making it easier for verification engineers to adopt modeling as part of their development process. The ability to automatically synthesize code and reverse engineer code back into UML allows engineers to refactor and elaborate source code without reducing the value of the models.

However, our current model-driven engineering process has a number of limitations that will provide the focus of our future work. Firstly, limited behavior is generated as part of the case studies undertaken during our evaluation process. The focus of the work was on investigating structural improvements through modularization. To be fully functional, the model needs to be extended to place more behavioral elements within the relevant structures. Secondly, our approach models a single testcase as a collection of constraints, however, we do not use Object Constraint Language (OCL) [OMG 2011a] as part of this modeling. OCL is a key component of the OMG standard for transforming models and its set of formal constraints and object query expressions would provide a more standardized approach to define these constraints than employed in our approach.

Finally, our initial research question was to investigate a profile specific for hardware verification and we focused our efforts on the e language. We would now like to more formally attach this work to the OMG’s testing standard by investigating our profile as an extension of the UML Testing Profile (UTP). This would more closely align our work with the testing standards.

7. SUMMARY
We have presented a novel model-driven engineering process and toolset that supports modularized development of hardware verification testbenches. Our process tackles...
the technical challenge of complexity which limits the reusability of testbenches as evidenced through an empirical analysis of testbench source code. Our toolset incorporates a new metamodel that facilitates the design of verification testbenches using high-level verification concepts, a model transformation tool that decreases the level of abstraction to produce an aspect-oriented source-code-level model of a testbench, a metamodel facilitating the modeling of the e hardware verification language, and a code synthesis tool.

We demonstrated through case studies that our model-driven development process and tools can have a positive impact on system modularity when applied to the reimplementation of an industry testbench. Through our modeling languages and transformation, we have facilitated the separation of functional verification concerns at design time that would have otherwise resulted in scattering and tangling in the testbench. These aspect-oriented techniques have been shown to improve the comprehensibility and reusability of testbenches, thereby reducing development time.

ACKNOWLEDGMENT
We wish to thank Darren Galpin, Infineon Technologies, Bristol Design Centre and Cristian Amitroaie and Cristian Slav, AMIQ Consulting for their help with this project. We also thank Aurélien Mariage, INSA Toulouse for his contribution to the development of many of the tools presented in the article.

REFERENCES
31:24 É. Linehan et al.


Received December 2010; revised October 2011; accepted December 2011