

Agile Formality: A “Mole” of Software Engineering Practices

Vieri del Bianco and Dragan Stosic
UCD CASL: Complex and Adaptive Systems Laboratory and
School of Computer Science and Informatics,
University College Dublin, Ireland
vieri.delbianco@ucd.ie and dragan.stosic@gmail.com

Joseph R. Kiniry
IT University of Copenhagen, Denmark
kiniry@itu.dk

Abstract:

Members of the agile programming and formal methods communities do not always see eye-to-eye. These two communities often do not talk to or learn from each other. Only recently, as highlighted by the September 2009 issue of IEEE Software, the IFIP workshop on balancing agility and formalism in software engineering, and the first edition of the international workshop for formal methods and agile methods, ideas from the two communities begun to be synthesized. While the problem-solving approaches and psychological attitudes of members of the two communities differ widely, we exploit this clash of viewpoints, exploiting a new development processes that actually blends, rather than mashes together, best practices from the two worlds. This paper summarizes our process and a supporting complex case study, showing that it is not only possible, but tasty, to combine the “chili pepper” of formal methods and the “chocolate” of agile programming, thus producing a tasty “Mole” (as in the highly-spices Mexican sauce) of software engineering practices.

1 Introduction

Agile and formal development methodologies usually do not blend together well. This is because of several reasons, the most important of which is often characterized as a radical difference in psychological attitudes about software development.

Formal methodologists often favour an in-depth, think-first approach, where the problem is understood, formalized, and solved; usually, but not always, adopting a waterfall-style of development. Once a formalization is developed, development and verification proceed, in an automated and interactive fashion. Consequently, projects that use formal methodologies (FMs) typically focus on critical systems with fixed requirements and somewhat flexible deadlines (“We’ll ship it when it is right.”).

Agile methodologists favour an highly incremental and iterative approach, specifically tailored to cope with changing requirements and precise deadlines. In projects that use agile methodologies (AMs), it is often the case that the problem is only partially understood. Thus, one focuses on the aspects of a system that must be implemented in the current development iteration. The solution initially developed is usually not optimal at all, but as it is refined through continuous code refactorings, its quality and validity improve over time. Test suites are used for several purposes, the most important of which are system verification, system documentation, and requirement specification, and they are strictly handmade. AMs are typically used in development settings where changing requirements and rapid delivery of the product are paramount.

The two worlds seem irreconcilable. Nevertheless, problems exist that would greatly benefit from both of them [BBB⁺09]. The problem class in which we have particular interest have *unstable requirements*, are *constrained by deadlines that cannot be postponed*, but also have *subsystems that must be formally specified and verified*.

1.1 Terminology

In this paper the IEEE 610 /citeJay1990 standard terminology for validation and verification is used. Paraphrasing the standard slightly, *verification* is the process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase, according to the requirements and design specifications (“you build the system in the right way”); *validation* is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements ensuring it meets the user’s needs (“you build the right system”).

Within the formal methods community, verification means something much stronger, and validation something quite different, than in the IEEE 610 standard. Loosely, within the FM community, *verification* (in the paper *formal verification*) means the use of formal methods to formally, statically (and usually modularly) verify that a system conforms to its specification (under all possible execution scenarios). On the other hand, *validation* (in the paper *formal validation*) means the use of traditional software engineering practices to check, statically or dynamically, that a system conforms to its specification (whether in the form of tests, requirements, etc.) by hand or via execution under some execution scenarios. In the present paper both perspectives are respected and acknowledged, with the attempt to be absolutely clear in using these terms unambiguously. Additionally, “...development of a system,” more precisely means “...analysis, design, development, verification, formal validation of a system”.

1.2 Case Study

Proposing new practices with no supporting evidence is an uninteresting proposition. A complex case study is mandatory to test bench the proposed practices. Our case study focuses on the development of a device driver to control and communicate with an embedded custom circuit board equipped with more than a dozen sensors. The communication protocol with the board is asynchronous and packet-based and the board is novel and custom-designed. The protocol is focused on controlling the board and reporting sensor measurements and board status.

This software driver has been developed using our process while satisfying seemingly contradictory requirements: (1) the protocol implementation must be thoroughly verified; (2) parts of the protocol must be formally specified; (3) a board simulator must be designed, implemented, and verified to match the specifications of, and actual behaviour of, the physical circuit board; (4) if board components change during development, the associated formal specifications and the simulator whose behavior reflects such changes must be updated accordingly; (5) strict software delivery deadlines. Consequently, these process requirements constitute an environment that benefits from both agile and formal approaches to software development.

A novel development process has been drafted to cope with the contrasting requirements of the case study, blending agile and formal practices. In the following the development process and the practices are summarized, and their successful application to the case study is reported, yielding several promising results.

1.3 Formal and Agile Integration

To enable the integration of the two worlds, the highly iterative and incremental approach found in most of the agile methodologies must be maintained, and formal validation methodologies, traditionally used in a waterfall development process (formally specify the system, implement the system, validate the implementation against the specification), must be adapted to a highly iterative one.

The most common software verification practices in agile methodologies must be considered, when trying to integrate formal and agile methodologies, since they are a fundamental part of the development process. The most significant verification practice is Test Driven Development (TDD) [Bec03], TDD is a software

development technique, originally defined in Extreme Programming (XP) [BA04] methodology, characterized by writing unit tests before writing the code. Actually, the general technique, consisting in writing the tests before the code, is used at high abstraction levels as well, since user stories (the requirements) are translated into test cases (the functional tests) before implementation. The test driven approach is a cornerstone of XP, but it is so popular that has been adopted in many other agile methodologies as well. It is also considered a good software development technique when used on its own, regardless of the enclosing development process.

The test driven approach relies on writing the tests before the actual implementation. It is applicable in a very small scale (Test Driven Development) yielding to activity cycles as short as a few minutes. It is applicable in a large scale as well (requirements immediately translated into functional tests), yielding to activity cycles as long as a full delivery iteration (usually no longer than a couple of months).

Some tentative attempts to reconcile formal and agile methodologies have been developed. The recurrent approach of the attempts consists in removing the handmade test suites, substituting them with different kinds of automated formal validation based on the system formal specification. The automated validation can be as simple as enabling runtime verification of assertions or automatically generating a test suite based on the system properties, or as complex as actually proving the whole system behaviour through static model and code checkers or theorem provers (formal verification).

Eleftherakis and Cowling in [EC03] propose XFun, an iterative and incremental process to generate X-Machines models, an extension of Finite State Machines. The development process impose a complete specification of the system, and its verification is done exclusively using the tools provided by the X-Machines formal methodology. The development process proposed in [HM03] by Herranz and Moreno-Navarro is quite similar: they propose an iterative process to model a system using SLAM-SL, an object-oriented specification language, and its tools. Some XP practices are fully adopted or considered compatible, as pair-programming, iterative and incremental development, system refactoring; while tests are completely replaced by the verification tools provided by the SLAM suite. A different formal methodology, but same approach, is found also in [SMSB05] where XFM methodology is explained.

As already mentioned, in all these attempts, the handmade test suites are automatically generated using the system formal specification, or completely replaced by theorem provers or static checkers. The requisite to enable these approaches is, at least, a specification of the entire system under development to enable simple verification techniques as runtime verification (obtained by assertions and design by contract [Mey97] preconditions, postconditions and invariants) and test suites automatic generation [CL02a, CL04, CKP05]. At most, a complete and sound system specification, usually supported by an appropriate formal language (to specify model and properties) and a constrained programming language, used to actually prove properties about the system under development [CH02, DNS05, KC05b].

The approach is simple: the verification, documentation and requirement specification purposes of the tests are moved on to the formal specifications, and a complete formal specification of the system under development is needed to apply these methodologies. The problem is that this requirement is hard to fulfil, since the effort required to write a complete formal specification of the system, in most real world complex cases, is usually greater than writing a suite of tests [Gla02]. If a complete formal specification is not built, all the previous methodologies share a common problem: the parts not formally specified are not verified at all. These can obviously be verified with the traditional methods, but the loops between development artifacts and related activities (code, tests, design, refactoring) are neither detailed nor enforced. The loops and relations between the different development artifacts are the inner engine of many agile methodologies, they impose an iterative and incremental pace to the development process.

The objectives we want to achieve are the following: to formally specify only parts of the system, in order to cope with constrained resources and unstable requirements; to draft an highly iterative and incremental development process that blends formal and agile practices.

The main specific problem, by now never addressed, is how to connect development artifacts of the two worlds together: we will define activity cycles, similar to what is found in agile methodologies (especially in XP), to solve this problem in an highly iterative development process. We will show how tests can drive the formal specification, how the formal specifications can drive tests and code development, how

handmade and automated tests can coexist and support each other, how the unspecified parts of the system can be incrementally specified. That is, we will show how to blend, and not compose, formal validation and agile verification practices: a “Mole” of verification practices¹.

2 A Real and Complex Case Study: Rapid Development in Small-scale Hardware Software Co-design

“Mole” practices are applied in the context of a real world case study that matches the problem requirements previously detailed: unstable requirements and development artifacts that need to be formally specified and validated.

2.1 The UCD CASL SenseTile System

The **UCD CASL SenseTile System** is a large-scale, general-purpose sensor system developed at the University College Dublin in Dublin, Ireland. The facility provides a scalable and extensible infrastructure for performing in-depth investigation into both the specific and general issues in large-scale sensor networking. This system integrates a sensor platform, a datastore, and a compute farm. The sensor platform is a custom-designed but inexpensive sensor platform (called the *SenseTile*) paired with general-purpose small-scale compute nodes, including everything from low-power PDAs to powerful touchscreen-based portable computers. Besides containing over a dozen sensors packaged on the *SenseTile* itself, the board is expandable as well, as new sensors are added to it easily.

The case study is focused in building the sensor board and its software driver concurrently. Because of hard time constraints and the initial unavailability of the custom board, we must progress concurrently with all the development tasks, including: (1) specification of the communication protocol; (2) specification of the physical sensor board; (3) development and fabrication of the physical board; (4) development of the embedded software for the board; (5) development of the communication protocol software driver of the board; and (6) development of software simulators of the board.

The development of the physical board, along with its embedded software (development tasks 2, 3, 4), is carried out by a third party under our guidance, thus it is not directly taken into account here. The specification of the communication protocol (development task 1) is a joint effort between ourselves and the third party, while the remaining tasks (5, 6) are performed in isolation. Dependencies are straightforward: the specification of the communication protocol (1) is the most stable element, but it still depends on the sensor board (2, 3, 4): large changes in the latter affect the former. Additionally, the driver and the simulator depends directly on the communication protocol.

Communication and synchronization with the external manufacturer is frequent but not optimal. The main synchronization artifact is the protocol specification, which remains stable at high-level, but is changed frequently in the low-level details. A common domain language has been found and agreed upon, but the formal methodologies and programming platforms differs widely. Consequently the informal specification of the communication protocol is the most reliable source of information.

2.1.1 Related Methodologies

There exist various approaches in literature addressing this kind of development constraints (rapid development in hardware software co-design), but all are focused on large-scale systems.

The hardware-software formal co-design methodologies usually have several common development artifacts [SDMH00, HKM01] including a high-level specification of the system, a translation (refinement)

¹A mole (*mōlā*) is a highly spiced Mexican sauce made chiefly from chili peppers and chocolate, served with meat.

of the high-level specification to low-level ones (hardware and software counterparts), the possibility to generate software code from the low-level specifications, a hardware simulator capable of simulating an hardware component based on hardware specification, the hardware component developed.

When considering the development of the device and of its software driver as separate entities that possibly have to be developed concurrently, the existing approaches are similar to the ones seen in the case of hardware-software co-design [Val95, SM02, RCKH09], all of which focus on a specification that aims to be as complete as possible.

A complete specification is neither feasible nor convenient in our case. The protocol specification must be enhanced incrementally and its complete specification cannot be provided. This makes the problem an ideal candidate to test our “Mole” of practices.

To perform this case study, an appropriate formal methodology must be chosen. It must be able to support a specification that is built incrementally. Consequently: formal methods that demand a complete specification must be avoided; additionally, to best synthesize with agile methods, the formal language must be able to automatically generate tests; finally, the formal method must be complemented by tools that support the verification of system properties both at design-time (via static analysis, for full functional verification of critical subsystems) and at runtime (for dynamic validation for integration with TDD).

2.2 The Chosen Formal Methodology: Formal Specifications with JML

Driven by these requirements and our expertise, the Java programming language and platform are chosen for the implementation, the Java Modeling Language (JML) [LBR01] is used to write formal specifications, and JML2 [LPC⁺04], and Extended Static Checker for Java (ESC/Java2) [CK, KC05a] tool suites are used as supporting tools [BCC⁺05]. The JML2 tool suite includes runtime assertion checker [CL02b] and a unit test generator [CL02a]. ESC/Java2 supports automatic, modular, static reasoning about the correctness of an implementation against its formal specification. These tool suites do not require a complete specification to be used effectively.

JML is the de facto standard specification language for describing Java programs behaviour. JML is a rich behavioural interface specification language (BISL) and thus focuses on the modular specification of classes and objects in which inheritance is used solely for behavioural subtyping [LBR99]. JML includes standard specification constructs like assumptions, assertions, axioms, and pre- and postconditions, framing clauses, and invariants. It also includes a rich model-based specification layer that includes mathematical models, data refinements, datagroups, and many other advanced object-oriented specification features [Cha06]. Many tools, ranging from compilers to static checkers to full-functional verification systems support the JML language [BCC⁺05].

JML is sometimes used in a Design-by-Contract style, where a specification is written from scratch, reusing existing APIs that have specifications of their own, and then an implementation is written conforming to that specification [Mey92]. The implementation is checked by whatever means appropriate using some subset of the aforementioned tools. At other times an existing piece of implemented and tested code is annotated with specifications after-the-fact, a process called Contracting-the-Design.

The two tools used most frequently to check the correctness of implementations are the JML Runtime Assertion Checker (RAC) and the Extended Static Checker for Java, version 2 (ESC/Java2) [CL02b, BCC⁺05, KC05b]. The former compiles executable assertions into runtime checks. The latter tool performs automated modular semantic static analysis to attempt to prove that the program under analysis does not misbehave (i.e., crash in a variety of ways) and conforms to its specification (lightweight functional correctness). As ESC/Java2 is fully automated, it is not sound, nor does it cover the whole Java and JML languages, and is thus not complete, but there is support for warning the user about both of these situations [KMD06].

3 Blending Formal And Agile Development

Agile methodologies are all based on a highly iterative and incremental process. They share a common approach on team management, customer relation, simplification and removal of unnecessary artifacts and activities, they rate working solutions and customers satisfaction as the most important indicators considered during development. The Agile Manifesto [BBvB⁺01] summarizes the philosophy and principles shared by all agile methodologies.

The Agile Manifesto does not suggest any specific development techniques, it describes agile methodologies from a very high abstraction level. Nevertheless, most agile methodologies share a similar approach to artifact verification. Functional test suites, used as precise requirements or user stories definition, are applied in DSDM [Sta97], XP and Crystal Clear [Coc04]. Unit tests and TDD are mentioned, and often completely accepted, in most of the agile methodologies defined so far.

High level test suites, composed of functional tests, describe requirements and bind together requirement documentation and system behaviour: an informal requirement is supported by a set of functional tests. The associated development cycle is one iteration long. Low level test suites, composed of unit tests, describe the module behaviour and bind together code documentation, code and the contract imposed on the module: unit tests represent the contract and the documentation. The associated development cycle is less than one day (as short as a few minutes).

Continuous refactoring [FBB⁺99] enables iterative and incremental development, since the system architecture and design, whether explicitly specified or not, must be cleaned up and modified, to accommodate new requirements, services and modules. Refactoring is possible and feasible because of the safety nets provided by the test suites, used as regression test suites. Automated test suites guarantees that the functionalities and behaviour already implemented in the code are not compromised by refactorings.

3.1 “Mole” practices

Our objective is to formally specify only parts of the system under development, in an incremental iterative process. Formal artifacts must coexist with informal ones: requirements, expressed as functional tests or as formal specifications, handmade and generated tests, documentation, source code. The development process needs to guarantee consistency over all the involved artifacts.

Both Test Driven Development (TDD) and classical Formal specification Driven Development (FDD) guarantee consistency on all the involved artifacts. Both TDD and FDD support iterative development. The development cycle adopted in TDD is shown in Figure 1 and the one adopted in classic FDD is shown in Figure 2. Unit tests used in TDD are substituted in FDD by formal specifications supported by the verification environment; in our case the verification environment is the generated unit test suite. Formal specifications replace unit tests: they specify, document and verify the system under development.

Is this enough to deal with formal and informal artifacts? FDD is used for the formal specified parts of the system, whether TDD is used for the remaining ones: a “composition” of formal and agile verification practices. Yet, it is not enough. There are several problems that still need to be addressed. No communication between the two worlds is permitted, it is not defined how to formally specify a module after it has been implemented and verified in a non formal way. A module only partially specified is a sort of hybrid that must be treated accordingly. Finally, in the FDD cycle shown in Figure 2, the implicit assumption is that it is always possible to identify the correct formal specifications of the system, but this is not always straightforward. Identify the correct formal specification is a difficult task, it has to be supported and verified by the development process.

A partially specified module cannot be verified properly only through automatically generated tests. Handmade tests have to be used together with generated ones. Handmade tests verify complex behaviours that are not formally specified, or complex behaviours that are formally specified but cannot be replicated by

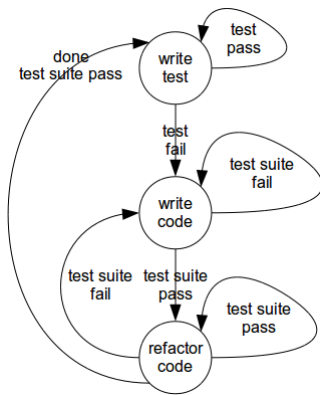


Figure 1: TDD development cycle.

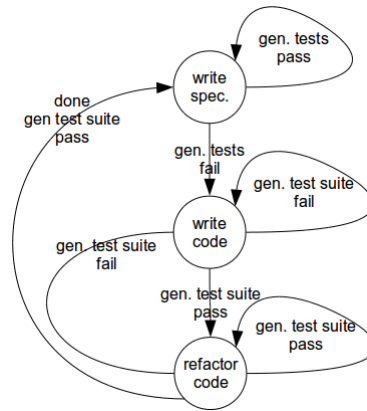


Figure 2: FDD classic development cycle.

the formal verification tools, because of the limits of the verification tools themselves. The constraints imposed by formal verification tools [Gla02] must be taken into account when deciding whether supporting the code with handmade tests, and verification tools based on automatically generated tests are not an exception. The development cycle able to blend formal specifications, handmade tests and code development is shown in Figure 3: the formal specification drives the handmade tests. Both handmade tests and the formal specification drive the code development.

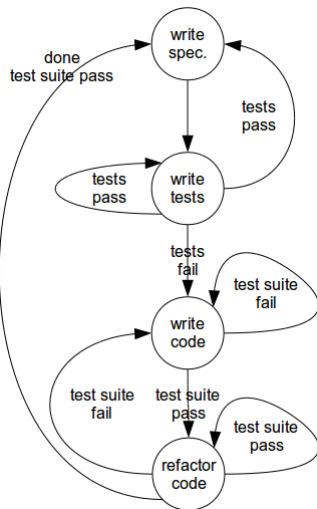


Figure 3: FDD development cycle with handmade tests.

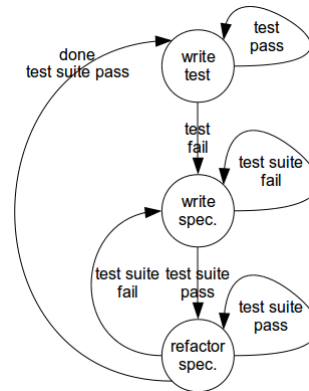


Figure 4: TDF development cycle.

The opposite path still needs to be covered, moving incrementally from a tested code to a formally specified code. In this case, new handmade tests and existing working code drives the formal specification, this is what we call Test Driven Formal specification (TDF). The resulting development cycle code is shown in Figure 4, TDF is used to incrementally add formal specifications to an existing working system.

When the “Mole” verification practices are used together the initial objectives are achieved: an incremental and iterative process, where development pace is defined by very short development cycles similar to TDD; the complete freedom to decide what is formally specified; the development cycles and the associated verification practices maintain the corresponding artifacts involved consistent, the correct development cycle is determined by the formality of the artifacts involved, ranging from pure development cycles, shown in Figure 1 and 2, to blended ones, shown in Figure 3 and 4.

4 “Mole” verification practices applied

4.1 Data stream and protocol description

The *Sensor Board* protocol is asynchronous and packet-based; the packets have a fixed length. The protocol is separated in multiple layers to ease its implementation in the driver, since mixing up different abstraction levels usually leads to overly complex and unmanageable code. The layers identified in protocol are the following: (1) packet byte structure: the internal representation of the packet; (2) packet info structure: the meaningful fields contained in the packets; (3) single packet rules: the content acceptable values, and how they influence each other; (4) packet sequence rules: the content acceptable values, based on values of previously received packets; (5) packet input output asynchronous rules: the content acceptable values and reaction constraints on output packets, based on previously transmitted input packets. For space reason, the first three layers are analyzed in the case study, considering only the output sensor data packet.

The packet byte and info structures of the output sensor data packet reflect the board capabilities and built in sensors. A single packet has to accommodate various types of data: fast, medium and slow data rate streams, together with metadata describing the sensors and the board state. The internal structure of the packet is strictly fixed, even if the data streams have different data rates. This choice originates from a trade-off between two constraints: the efficiency of the transmission (redundant data to be minimized) and the complexity to build a packet (complexity to be minimized, since the processing capabilities in the *SenseTile Sensor Board* are limited).

A packet is internally composed by 82 frames. A frame accommodates data from the fast and medium data rate streams (there are 4 fast data rate stream and 8 medium data rate stream channels) and their associated metadata. Fast, medium and slow data rates have not a common multiple; a packet, capable to accommodate outputs at different rates, in a fixed structure, relies in two forms of redundancies, for slow and medium data rates:

- Slow data rates: slow data is located at the beginning of each packet, the data is simply repeated (it is equal to the previous packet in the stream) when no new data is available.
- Medium data rates: the 8 medium rate channels are multiplexed together into the medium data rate data sample of each frame. The medium data rate sample channel is a piece of info extracted from the frame description (frame metadata), including the possibility to have no data at all. E.g. a frame contain a medium data rate sample from channel 0, the following frame a sample from channel 5, the following frame no sample at all, etc.

The single packet rules delimit the boundaries of the values obtainable from the packet: each defined sensor represented in the packet, as well as the metadata describing the *SenseTile Sensor Board* and the packet and frame contents, are constrained by a range of acceptable values. There are also rules affecting more than one value (i.e. if a specific sensor is active the reading value must be in a defined range), and rules specifying a correct sequence of frames (i.e. only one switch in sampling rate is allowed in the frame sequence of a single packet).

4.2 Data stream and protocol specification

The specifications of the protocol are distilled and refined incrementally. The protocol is divided in various (thin) layers, and each of the layer is verified with a different approach: some of the layers are specified formally.

Packet byte structure specification Verification is obtained through an handmade test suite, composed of unit and integration tests; the tests specifies the behaviours of the implementation, which is capable of recognizing the proper packet structure in a binary data stream. The implementation is tricky: it searches

for repeated patterns into a byte stream to be sufficiently confident on how the stream is decomposed into packets. Besides, the packet byte structure is not stable: the internal byte structure changed several times during development. Therefore, packet byte structure is not formally specified.

The test suites are used to verify board simulators (low-level), but they cannot be used to check properties at runtime. Code and tests are implemented using TDD.

Packet info structure specification Verification is provided by handmade test suite, JML annotations and generated unit test suite. It has been built starting from handmade tests only (TDD development cycles). Later on JML annotations have been added incrementally, using a mix of “Mole” practices: to move from handmade tests to formal specifications, TDF have been initially used; then all the “Mole” practices have been used according to the artifacts involved.

The test suites are used to verify board simulators (high-level), RAC and JML annotations are used to check the formally specified properties at runtime. Code and tests have been initially implemented using TDD, then using all the “Mole” verification practices.

Single packet rules The verification is provided exclusively by the formal specification with JML language, combined with the generated unit test suite and handmade test suite to verify the most complex behaviour. It is built starting from JML annotations, supported by handmade tests when needed.

The test suites are used to verify board simulators (high-level), RAC and JML annotations are used to check most of the rules at runtime. Code and tests are implemented using both forms of FDD.

4.3 On simulators

Simulators are used to test parts of the system minimizing dependencies: a simulator can be used by upper layers, with no need to provide the functionalities of the lower layers. The simulators proved essential during the development, since the physical board was not available.

Theoretically a separate simulator is needed for each protocol layer to specify, develop and test each layer in isolation. But, because of limited resources and effort available, the layer structure is not matched by corresponding simulators. Two simulators are available, they have been built concurrently with the specification, the driver implementation, and the tests.

The driver API is splitted in two abstraction layers: the general high-level interface, that exposes the main functionalities of the board and the main contents of the packets, and the lower level implementation that parses the data streams in and out the board, meant to translate the higher level instructions and data in properly formed packets. The simulators are built according to these two abstraction layers. The high-level simulator implements the interfaces providing the methods to deal with an abstracted *Sensor Board*: the simulator knows nothing about the details of the real format of the binary data streams. The low-level simulator is capable to rebuild the *Sensor Board* data streams: the in and out data streams are built exactly as the sensor board is expected to parse or generate.

4.4 JML specification examples

The JML specifications are of varying complexities. Some of them are rather simple, focusing on constraints that should hold when calling a method (the preconditions) and constraints on the return value (the very basic form of postconditions). In listing 1 a simple JML specification example is shown, the method `getTemperature` is declared `/*@ pure */`, which means that it cannot change the state of any object (a postcondition); the specification also constraints the return value with a lower and upper bound

Listing 1: A simple specification with JML annotation: simple postconditions.

```
/*@
    ensures \result >= -880;
    ensures \result <= 2047;
@*/
/*@ pure @*/ short getTemperature ();
```

(another postcondition).

The complex specifications usually focus on properties regarding the behaviour of a whole object. In listing 2 a complex invariant example is shown; an invariant is a property maintained during the life cycle of an object, more precisely, an invariant is assumed on entry and guaranteed on exit of each method of an object. The invariant is constraining the number of samples for each medium data rate streams: the total number of streams is `Frame.ADC_CHANNELS`, the total number of frames is `FRAMES`, the constant constraining the number of samples is `FRAMES/Frame.ADC_CHANNELS+1`, meaning that the samples contained in a frame are fairly distributed on the channels. The valid samples are counted parsing all the frames contained in a packet, selecting only the matching valid samples. A medium data rate stream sample is considered valid when `isADCActive()` method returns **true**.

Listing 2: A specification with JML annotation: a complex object invariant which constraints the number of samples for each medium data rate stream channel.

```
/*@
    invariant (
        \forall int channel;
            0 <= channel &&
            channel < Frame.ADC_CHANNELS; (
                \num_of int i;
                    0 <= i &&
                    i < (FRAMES-1); (
                        getFrame(i).isADCActive() &&
                        getFrame(i).getADCChannel() == channel
                    )
                ) <= (FRAMES / Frame.ADC_CHANNELS + 1)
            );
@*/
```

4.5 Test cases

The unit test cases that verify the protocol driver are of two kinds: handmade unit tests and automatically generated unit tests based on JML specifications; they are complementary and built to be used together.

The test effectiveness is evaluated for each test suite; the evaluation is carried out in section 4.7. The test effectiveness evaluation considers three elements: effective results on piloting the real board (quantitative), code coverage (quantitative), development help and usefulness (qualitative).

Handmade tests The package structure of the driver is shown in Figure 5: two independent packages are defined (`Stream` and `Driver`). The packages are abstract, that is, they mainly contain abstractions; in Java language this is translated into a package which contains mainly abstract classes or interfaces.

The dependency between the two abstract packages is not direct, it is realized through an implementation (`StreamDriver`). This is needed to maintain a high decoupling of the packages, and is the result of applying the dependency inversion principle [Mar96].

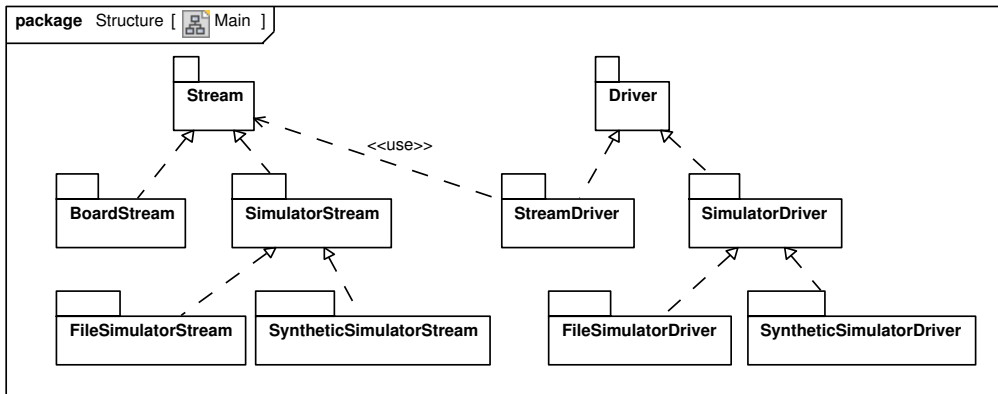


Figure 5: Main packages class diagram.

The test package structure, shown in Figure 6, reflects and mimics the code structure; the tests for an abstract package are abstract, and implemented by the package that tests a corresponding system implementation. This is a well known test pattern (the Abstract Test Pattern [Tho04]) used to test that the contracts defined in the abstractions are respected in all the implementations. For instance, package `DriverT` contains abstract tests for the abstractions of package `Driver`, package `StreamDriverT` inherits the abstractions of `DriverT` and makes them concrete, to test the corresponding implementation `StreamDriver`; package `StreamDriverT` also contains stand alone tests written specifically for the implementation `StreamDriver`.

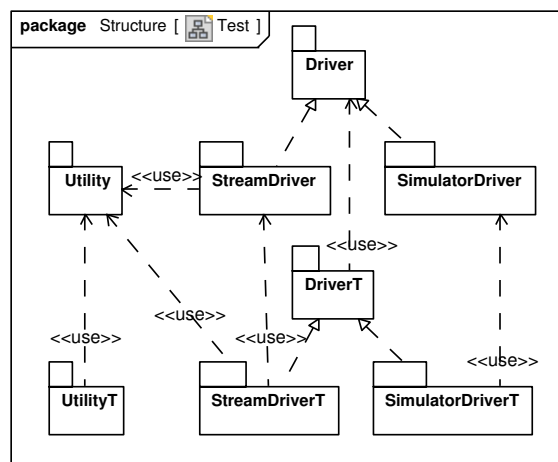


Figure 6: Test packages class diagram.

Generated tests The JML specifications are used to generate tests. The resulting test package structure is shown in Figure 7. The package dependencies are defined by the specifications; for instance, `SimulatorDriverST` are generated tests, they test the implementation `SimulatorDriver`, and are

generated using the corresponding specification `SimulatorDriverS`. The generated tests do not maintain the inheritance structure of the code and the specification.

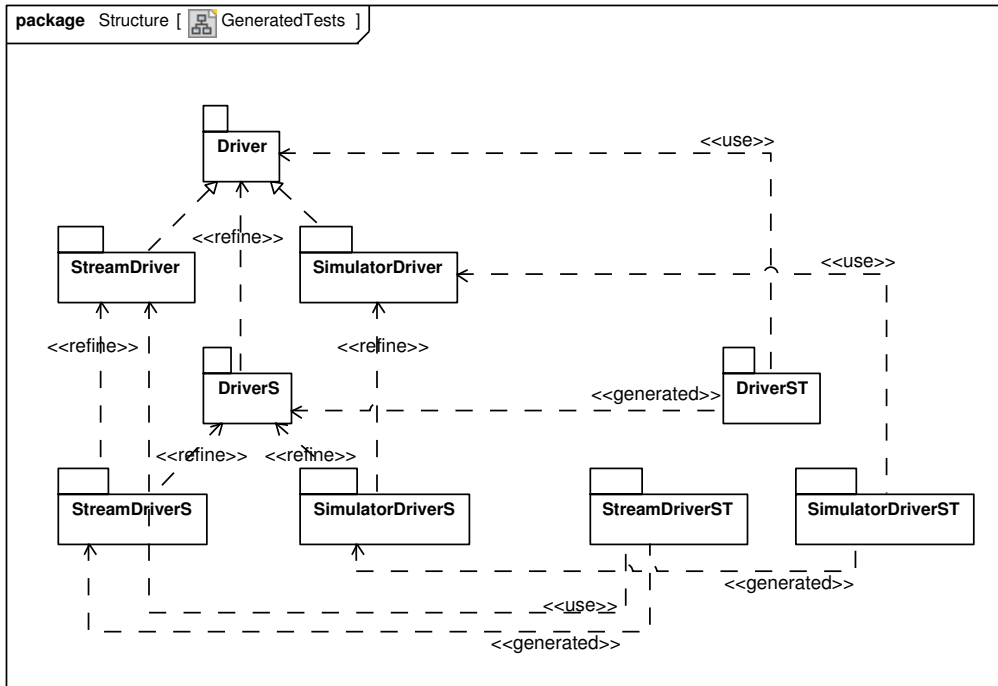


Figure 7: Generated test packages class diagram.

On simulators, again The simulators are adopted in the handmade test suits as stubs. The resulting structure is shown in Figure 8. For instance, in Figure 8 the test suite `StreamDriverT` is using the `SimulatorStream` to properly simulate the `Stream` parsed by `StreamDriver`, which is the system under test.

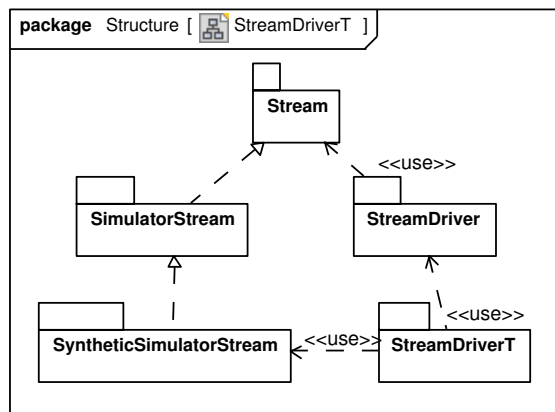


Figure 8: `StreamDriver` test packages class diagram: use of the stream simulator.

Simulators were essential to test the *SenseTile Sensor Board* driver, since the physical board was not

available. The simulators are adopted in both the handmade and the generated test suite; both the suites need a proper set of stubs in order to be executed.

4.6 Test cases on the job

The handmade test suite is composed by over 140 tests, while the generated one is 100 times bigger, totalling over 14000 tests. The large number of generated tests are explained looking closely at how the generated tests are created by the JML framework. The generated tests explore the possible input combinations on public methods, combining the type data ranges that are specified for the test suite (see [CL02c] for more details on how the JML framework works on generating unit tests). For instance, let's suppose a method `m(int p1, int p2)` must be tested in a class `C`; the data ranges specified for type `int` are `{1, 2, 3}` and for type `C` are the instances `{c1, c2, c3, c4}`. The JML framework generates $24 = 3 \times 3 \times 4$ tests, since they are the combination of the data ranges involved (the data range of the parameters and the data range of the type owning the method under test). Nevertheless, not all the generated tests are executed: a generated test is not executed, unless the preconditions of the called method are satisfied. Thus, depending on the preconditions specified for a method, the number of the generated tests that are actually executed in the test suite are significantly smaller compared to the total number of generated tests.

Test suites execution time are different: the handmade test suite, with the runtime assertion checking active, runs completely in less than one minute on an average PC, while the generated test suite runs in more than 20 minutes (the rough ratio is 1 : 60). Most of the time is spent on setting up the suite; only considering the real execution of the tests, with no suite setup, the execution time reduces to less than 4 minutes (the rough ratio is 1 : 10).

Code coverage metrics are used to compare in a quantitative way the effectiveness of the test suites. This is only an indicator, since it is accepted that code coverage alone cannot assess the quality of a test suite [Mar99, CKV06]. Various coverage metrics exist, the test suites are analyzed with statement code coverage (one of the simplest types)²: statement coverage reports whether each statement is encountered during execution. Statement code coverage is easy to implement, but lacks the capability to explore the control flow graph associated to the code, that is, decision branches are not explored. The coverage results are reported in the Table 1.

Table 1: Statement code coverage result, categorized by source package.

	<i>hand</i>	<i>generated</i>	<i>total</i>
<i>Driver</i>	15.9%	6.3%	15.9%
<i>StreamDriver</i>	76%	79.7%	88.5%
<i>SimulatorDriver</i>	64.9%	44.2%	71.2%
<i>Utility</i>	98.1%	74.5%	98.1%

4.7 Retrospective on test cases effectiveness

The coverage measures reported in Table 1 do not reach 100%. This is because of the effects of both non public utility methods, and abstract methods in interfaces. The effect of non public methods (package Java visibility) is that these methods are taken into account as public and protected ones are, during code coverage calculation; these methods are not part of the interface, the system does not depend on them,

²The tool used to obtain statement code coverage metrics is [Emma](#).

they are only used in object initialization or in object setups performed during unit tests. In fact there are package and private code blocks that are not reachable by construction from public and protected methods, and this is a precise characteristic reflecting the code architecture, meant to ease class instances setup and configuration for testing purposes. The used statement coverage tool is not able to distinguish these code blocks, so this effect is unavoidable. The effect is mainly seen in low `SimulatorDriver` code coverage figures: the `SimulatorDriver` package has many utility non public methods.

Regarding interfaces, an interface contains no statements, so when an interface method is called, it is the method of the implementation class used that is actually covered. This effect is visible in `Driver` coverage figures: `Driver` package contains interfaces (that are not counted at all) and some very simple real class (exceptions) that are not thoroughly tested. The net result is extremely low statement code coverage figures because the very simple classes are the only ones that are actually covered.

The statement code coverage figures show that the test suites do a good job on the most important package, the `StreamDriver`. The generated test suite reaches almost 80%, while the handmade tests have only a slightly lower coverage ratio: 76%. The combination of test suites raise the coverage ratio to 88.5%. This is a clear indication that one test suite is not completely overlapping the other: the intuition that test suites are not alternatives, but have to be used together to achieve the higher benefits, is confirmed. A similar indication is provided by `SimulatorDriver` package as well.

The qualitative difference of the two test suites are understood observing how the suites are built. A handmade unit test usually requires objects initialization (it involves several method calls), the call to the method under test, and the following assertions to verify the expected state change on the objects involved. A generated unit test has a more focused and limited scope: only one method call is performed in each test, therefore it is difficult to explore complex behaviours. With a complete specification, and specifying large enough data ranges, the generated tests achieve the same expressive power as the handmade ones; but the specification is not complete, and a large enough data ranges is long to develop. The test suites give their best when used in a combined approach: the generated test suite, checking simpler and formalized behaviours, and the handmade, checking the more complex ones, whether formalized or not.

Effectiveness on board delivery The first *SenseTile Sensor Board* prototype builds packets with no data, but the packets are built in a way not consistent with the specification. The board errors are detected by simply using the driver implementation, the `StreamDriver` package.

The second prototype builds packets with values taken from real sensors installed on the board. The only sensors that are not working are the sensors devoted to the fast rate data streams (audio sensors). One error needed to be corrected in the driver implementation, because of a (rare) combinations of conditions not initially covered by the test cases, but that occasionally showed up during real use. One error was discovered in the low-level system driver used by the developed driver (the system driver is outside our development scope): to eliminate the error, an upgrade of the low-level driver to a new beta version was necessary. No other errors were detected on the driver side.

The second prototype respected the packet byte structure specification, but was not fully compliant to the packet info structure specification and the single packet rules. The formal JML specifications and the runtime assertion checker correctly identified these errors. 4 protocol errors regarding the packet info structure specification, and 2 protocol errors regarding the single packet rules were found, and are expected to be corrected in the following release. No other errors, except the mentioned ones, have been discovered up to now.

5 Conclusions

In this paper we focus on a specific set of development problems that neither agile methodologies nor formal methodologies are completely comfortable to cope with. The problems we are interested in are characterized by unstable requirements and artifacts which must be formally specified and verified, and are

constrained by deadlines that cannot be postponed. We propose a blend of agile and formal engineering practices, enclosed by an iterative and incremental development process: the “Mole” development process.

The verification practices proposed recreate the fast feedback development environment we find in Test Driven Development, used with formal and non formal artifacts. A total of four practices are presented, two of them are conservative, following closely the development practices of agile and formal methodologies, while the other pair are innovative, blending together elements from both worlds.

The practices have been applied successfully to develop a driver for a custom embedded sensor board, equipped with board simulators and protocol verifiers. The protocol and the board specifications were informal and unstable. Yet a working software product was needed as soon as possible, with incrementally added functionality, and keeping it linked with the last specifications of the protocol and the board. “Mole” verification practices enabled us to keep in sink formal specifications (JML annotations), informal specifications (test suites) and source code, through very fast and short development cycles. We developed handmade test suites, generated test suites and JML annotations, which successfully supported the verification of the driver and the simulators, and the board as well, when it was delivered.

The practices are very general, and are applicable to different development languages and formal methodologies. One important constraint is on the formal methodology, that must be able to cope with partial specifications; another constraint is on the tools available, that must be able to verify the system properties at runtime and partially automate the verification process of the formalized properties.

References

- [BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, second edition, 2004.
- [BBB⁺09] Sue Black, Paul P. Boca, Jonathan P. Bowen, Jason Gorman, and Mike Hinchey. Formal Versus Agile: Survival of the Fittest. *Computer*, 42(9):37–45, 2009.
- [BBvB⁺01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. *Manifesto for Agile Software Development*, 2001.
- [BCC⁺05] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, February 2005.
- [Bec03] Kent Beck. *Test-driven development: By example*. Addison-Wesley, 2003.
- [CH02] N. Cataño and M. Huisman. Formal Specification of Gemplus’ Electronic Purse Case Study using ESC/Java. In *Proceedings of Formal Methods Europe (FME) 2002*, number 2391 in Lecture Notes in Computer Science, pages 272–289. Springer-Verlag, 2002.
- [Cha06] Patrice Chalin. Early Detection of JML Specification Errors using ESC/Java2. In *Proceedings of the Workshop on the Specification and Verification of Component-Based Systems (SAVCBS)*. ACM Press, November 2006.
- [CK] David R. Cok and Joseph R. Kiniry. The Extended Static Checker for Java, version 2, 2003–. See <http://kind.ucd.ie/products/opensource/ESCJava2/>.
- [CKP05] Yoonsik Cheon, Myoung Yee Kim, and Ashaveena Perum. A complete automation of unit testing for Java programs. In *Proceedings of the 2005 international conference on Software Engineering Research and Practice*, pages 290–295, 2005.
- [CKV06] Hana Chockler, Orna Kupferman, and Moshe Vardi. Coverage metrics for formal verification. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4):373–386, 2006.
- [CL02a] Yoonsik Cheon and Gary Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *ECOOP 2002 à Object-Oriented Programming*, pages 1789–1901. 2002.

- [CL02b] Yoonsik Cheon and Gary T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *the International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, June 2002.
- [CL02c] Yoonsik Cheon and Gary T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In Boris Magnusson, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255. Springer-Verlag, June 2002.
- [CL04] Yoonsik Cheon and Gary T Leavens. The JML and JUnit Way of Unit Testing and its Implementation. 2004.
- [Coc04] Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley, 2004.
- [DNS05] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the Association of Computing Machinery*, 52(3):365–473, 2005.
- [EC03] G. Eleftherakis and A. J Cowling. An Agile Formal Development Methodology. In *1st South Eastern European workshop on Formal Methods (SEEFM 03)*, page 36â47, 2003.
- [FBB⁺99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, June 1999.
- [Gla02] Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, October 2002.
- [HKM01] A. Hoffman, T. Kogel, and H. Meyr. A framework for fast hardware-software co-simulation. In *Proceedings of the conference on Design, automation and test in Europe*, pages 760–765, Munich, Germany, 2001. IEEE Press.
- [KC05a] Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML. In *Proceeding of the International Workshop on the Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS) 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2005.
- [KC05b] Joseph R. Kiniry and David R. Cok. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Proceeding of the International Workshop on the Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS) 2004*, volume 3362 of *Lecture Notes in Computer Science*. Springer-Verlag, January 2005.
- [KMD06] Joseph R. Kiniry, Alan Morkan, and Barry Denby. Soundness and Completeness Warnings in ESC/Java2. In *Proceedings of Specification and Verification of Component-based Software (SAVCBS)*, Portland, OR, 2006.
- [LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer Academic Publishing, 1999.
- [LBR01] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, August 2001.
- [LPC⁺04] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, and Joseph Kiniry. *JML Reference Manual*. Department of Computer Science, Iowa State University, 226 Atanasoff Hall, draft revision 1.94 edition, 2004.
- [Mar96] Robert Cecil Martin. The dependency inversion principle. *C++ Report*, 8(6):61–66, 1996.
- [Mar99] Brian Marick. How to Misuse Code Coverage. In *Proceedings of the 16th international conference on Testing Computer Software*, pages 16—18, 1999.
- [Mey92] Bertrand Meyer. Applying Design by Contract. *IEEE Computer*, 25(10):40–51, October 1992.
- [Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, second edition, 1997.
- [RCKH09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: taming device drivers. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 275–288, Nuremberg, Germany, 2009. ACM.

- [SDMH00] F. Slomka, M. Dorfel, R. Munzenberger, and R. Hofmann. Hardware/software codesign and rapid prototyping of embedded systems. *Design & Test of Computers, IEEE*, 17(2):28–38, 2000.
- [SM02] R. Siegmund and D. Muller. Automatic synthesis of communication controller hardware from protocol specifications. *Design & Test of Computers, IEEE*, 19(4):84–95, 2002.
- [SMSB05] Syed M. Suhaib, Deepak A. Mathaikutty, Sandeep K. Shukla, and David Berner. XFM: An incremental methodology for developing formal models. *ACM Trans. Des. Autom. Electron. Syst.*, 10(4):589–609, 2005.
- [Sta97] J. Stapleton. *Dynamic Systems Development Method*. Addison Wesley, 1997.
- [Tho04] J. Thomas. *Java testing patterns*. John Wiley & Sons, 2004.
- [Val95] A. Changuel C. A. Valderrama. A unified model for co-simulation and co-synthesis of mixed hardware/software systems. <http://www2.computer.org/portal/web/csdl/doi/10.1109/EDTC.1995.470395>, March 1995.
- [HM03] ngel Herranz and Juan Moreno-Navarro. Formal Extreme (and Extremely Formal) Programming. In *Extreme Programming and Agile Processes in Software Engineering*, page 1012. 2003.