Submitted to 2014 SERP - FDPTA: LATE BREAKING PAPER

Formal Specification-Driven Development

Richard Rutledge
College of Computing
Georgia Institute of Technology
Atlanta, GA
rrutledge@gatech.edu

Sheryl Duggins, Dan Lo, Frank Tsui
Department of CS and Software Engineering
Southern Polytechnic State University
Marietta, GA
{ sduggins, clo, ftsui }@spsu.edu

***Abstract*— Since the inception of software engineering we have focused on product quality and the process that was needed to develop safety critical products. The definitions of quality and associated attributes grew as software engineering matured. One of the early "silver bullets" was the hope that incorporating formalism in the process would bring us marked quality improvement, but the drawback of extensive training and competency detracted from that approach. Nevertheless, recently we have made significant progress in this area, especially in the form of improved tools and training. This paper examines some of the recent improvements made in processes such as Test-Driven Development (TDD), Behavior-Driven Development (BDD), and associated automation tools. It proposes a related methodology, a Formal Specification-Driven Development (FSDD) process that embraces these recent improvements. The paper recognizes that in the area of knowledge transfer, humans will always make errors, however, FSDD will lessen those errors through improved unambiguity, correctness, completeness, and consistency.**

Keywords-component; formal methods; test-driven development; formal specification; behavior-driven development;

I. INTRODUCTION

In search of productivity and quality gains, some software engineers have modified the traditional software development process model. Early work stressed evolutionary, incremental improvement. Efforts such as the Software Engineering Institute (SEI) Capability Maturity Model (CMM) and Software Process Improvement and Capability Determination (SPICE/ISO 15504) encouraged increasingly heavyweight process models. In counter-point, proponents of agile software development often promote revolutionary, lightweight process models. These models may prune sub-processes and/or fundamentally reorder the remaining sub-processes. This paper will examine some of the advantages and shortcomings of one such approach, Test-Driven Development (TDD) [1] and a derivative, Behavior-Driven Development (BDD) [2]. The paper will then introduce a new artifact to the traditional process model and present an argument that this traditional-derived model delivers the advantages of TDD and BDD without their shortcomings. The new artifact is a formal design specification expressed in a behavioral specification language. Hence, the process model proposed herein is referred to as Formal Specification-Driven Development (FSDD).

The scope of this paper will be restricted to a qualitative argument presenting the advantages of a FSDD approach. It should be viewed as a proposed methodology leading to future research to establish the quantitative efficacy of FSDD. Attaining the quantitative efficacy of a process model is difficult due to a number of factors. Consider TDD: although introduced in 2003, TDD still lacks such quantifiable justification. Tsui defines a set of 5 items or criteria that should be included in a process definition: i) activities, ii) control, iii) artifacts, iv) resources and v) tools [3]. Here, we will examine the merits of FSDD mainly through the perspective of Tsui's item iii) and item v), the artifact and the tool.

Analogously to TDD/BDD, the practical application of FSDD requires considerable automation and tool support. Since TDD requires the frequent execution of all test cases by the developer, the cases must be run quickly and efficiently. Hence, the test suite must be fully automated. Similarly, FSDD requires the formal design specification to be utilized in both the implementation and test phases. Although it is possible to perform these steps manually, specification without automation would probably be impractical and would abrogate many of the advantages cited in this paper.

This paper will demonstrate why new methods of software development are needed and provide the underlying motivation. It will establish the viability of the tool and automation support presumed above. It will describe TDD and BDD and introduce FSDD and work through a process example using FSDD. Finally the paper will suggest future work in this area.

II. MOTIVATION

Computer software plays an essential and critical role in managing the infrastructure of a modern society. It is integral to the operation of nuclear power plants, household toasters, and all manner of environments in between. When software fails, airplanes do not fly and cars do not drive. Yet the challenge of building reliable software remains even as the size of software projects scales ever larger. Even the ubiquitous cell phone contains about five million lines of code [4]. Under research funded by the Department of Homeland Security, Chelf at Coverity examined 32 well established, open-source software projects encompassing 17.5 million Lines of Code (LOC) [5]. They calculated an average defect density of .434 per thousand lines of code. In addition to the risk to safety, software defects are also expensive. In 2002 the National Institute of Standards and Technology (NIST) estimated that software defects cost the US economy over $60 billion each year. Further, NIST also determined this figure could be reduced by $22 billion if these defects could have been found more efficiently.

In response to the current state of software quality, the Verified Software Initiative (VSI) was founded to "gain deep theoretical insights into the nature of correct software construction, to radically advance the power of automated tools for the construction and verification of software, and to benchmark the capabilities of such tools through convincing experiments [6].” The VSI is a 15-year collaborative research project with ambitious goals toward an ultimate objective of defect free software. These goals include both the establishment of the theoretical foundations of software verification and the development of a sufficient toolset to validate the approach with real-world software.

While formal specification languages, such as Z, could be employed to demonstrate program correctness, they are still unwieldy, impractical, and cost prohibitive for the vast majority of projects. In current industry practice, an implementer is given a module design specification consisting of Unified Modeling Language (UML) and natural language. The implementer then develops both the source code and the unit tests for validation. However, this process can be substantially flawed. If the implementer misinterprets the design specification, that error will be reflected in both the code and the unit tests, and is likely to remain undetected at least until module integration, if not even later. The VSI effort is one promising attempt to resolve this problem.

A formal specification language is a mathematically precise notation for stating the properties of a software component. A behavioral specification language is a type of formal language intended to express the behavior of a module. A behavioral specification language similar to Java Modeling Language (JML), Spec#, Vienna Development Method Specification Language (VDM-SL), or the Object Constraint Language (OCL) can be employed to mitigate this process flaw. With sufficient tool support, a behavioral design specification can be used for both dynamic, run-time analysis and static analysis of an implementation. Much of the current implementer written unit tests could be replaced with automated specification driven testing, combining the advantages of Bertrand Meyer’s Design by Contract [7] with Test Driven Development. Additional tools could support steps in the Software Development Life Cycle (SDLC) beyond implementation. For example, specifications could be scanned for boundary value conditions [8] to aid the quality assurance team in test case generation.

An ideal behavioral specification language would be rigorous, expressive, assessable, executable and abstract. However, note that aspects of these attributes conflict. Otherwise, designers would long since have adopted Z or a derivative as the specification language of choice. An ideal specification language must discern an appropriate blend.

III. TRADITIONAL SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

A. Description

Traditional software development process methodologies recognize a common sub-set of distinct activities. These include, among others, requirements analysis, software design, implementation, and test. Methodologies differ in how these activities are organized and performed. At the completion of each element of an activity partition, an artifact must be produced to transfer project specific knowledge to the next agent in the selected process. Thus the artifact can be seen as a project-specific bridge between knowledge domains. Both the provider and the consumer of an artifact must be able to comprehend it, and the extent to which they each understand the artifact identically has a significant effect on the fidelity of the next activity performed. Since this is an iterative cycle, such transfer comprehension errors accumulate. The nature of the artifacts and methods of knowledge transfer will be examined more closely below.

B. Issues

Formal inspections and reviews of artifacts in software engineering to detect and remove defects is a natural part of the software process for large and complex projects today. However, an error in knowledge transfer is unlikely to be detected early in the process so the knowledge transfer error may be propagated indefinitely.

The possibility of error exists whenever an artifact is transferred to another agent. In order to minimize knowledge transfer errors, artifacts should possess a subset of the characteristics Pfleeger has identified for one particular artifact, a software requirement [9]. This list includes correct, consistent, unambiguous, and complete. Additionally, these characteristics must apply equally in both the producer and consumer knowledge domains. Considering the traditional SDLC, note that agent re-interpretation of artifacts occurs throughout the model. Thus a unit test can only detect an activity error and will be oblivious to knowledge transfer errors.

IV. BEHAVIORAL/TEST DRIVEN DEVELOPMENT

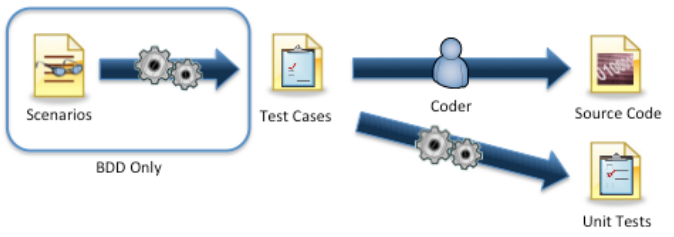

Figure 1. TDD/BDD Artifact Production

A. Description

1. Test-Driven Development (TDD)

Programmers construct unit tests to exercise the proposed unit of source code before releasing as completed functionality. In 2000, Kent Beck introduced eXtreme Programming (XP), an agile development methodology [10]. One of the key practices of XP requires the programmer to build the unit tests first, before any source code is written. Beck later generalized this practice as Test-Driven Development (TDD) [1] and summarized the practice as repeated iterations of the following:

1. Write a new test case.
2. Run all the test cases and see the new one fail.
3. Write just enough code to make the test pass.
4. Re-run the test cases and see them all pass.
5. Refactor code to remove duplication.

For practical usage, the frequent application of this procedure implies the full automation of unit test performance. Since passing all unit tests is synonymous with code completion, the collective set of unit tests **is** the design specification. As such, the unit tests serve both verification and validation. Thus TDD enhances unambiguity and completeness as compared to a traditional model.

One clear advantage to TDD is that it provides an unambiguous design specification. Thus the source code either conforms to its specification, or it does not. In 2004, Erdogmus et al conducted an empirical study of TDD (referred to as Test-

First) [11] and found a small productivity gain. This gain was attributed to better task understanding, better task focus, faster learning, and lower rework effort.

2. *Behavior-Driven Development (BDD)*

While teaching agile practices to industry, Dan North noticed a few recurrent problems with TDD [12]. "Programmers wanted to know where to start, what to test and what not to test, how much to test in one go, what to call their tests, and how to understand why a test fails". Some of these questions arose because TDD inverted their standard practice. They knew what they wanted to code, and then tested what they wrote. TDD requires them to first decide what they want to test. Other questions resulted because Beck described TDD in general terms rather than a procedure. North responded with Behavior-Driven Development (BDD). It is a process implementation of the TDD approach. Hence it possesses the TDD advantages, while addressing what North perceived as its shortcomings. In order for unit testing to drive the implementation process, the tests must constitute a behavioral specification. Supporting the terminology shift, North derived his automation test tool, JBehavior, from Java's JUnit.

North's next significant insight from the phraseology shift was that the semantics of the unit tests (behaviors) was now accessible outside the programming staff. When BDD's naming conventions are followed, a straightforward processing of the unit tests produces a set of English sentences accessible to an analyst. The final step to current BDD practice is provision for a "ubiquitous language" that can be read and written by analysts, and read by frameworks such as JBehavior. These goals are realized by projects such as rSpec and Cucumber for the Ruby language [2]. By extending applicability back into the analysis domain, BDD yields additional completeness and some consistency improvements over TDD.

B. *Issues*

Several issues with TDD/BDD can be qualitatively considered and have been quantitatively analyzed. This paper will discuss the specific first (BDD), and then proceed to the general (TDD). Since the collection of scenarios **is** the specification, the analyst is constrained to a language provided for by the developer and the analyst may only make pre-arranged lexical changes to the scenario specification. These limitations are in stark contrast to the goal of a ubiquitous language. Thus the process is exposed to the same knowledge transfer errors as before and the observed process improvement with regard to consistency is severely restricted.

Without the expansion into the analysis and test realms, BDD devolves into TDD with a specific vocabulary. BDD activities are limited to the developer who specified behaviors (unit tests) before writing any implementation code. In 2005, Erdogmus et al performed one of the first empirical studies of TDD [11]. In a study involving thirty-five third-year students, he found no impact on quality and a statistically insignificant improvement in productivity. Erdogmus explained this lack of clear results in terms of the limited scope of the projects under test and the inexperience of the participants. However, another explanation is that TDD does not address knowledge transfer errors, and thus is not a major contributor to product quality. In 2010, Kollanus reviewed forty empirical studies of TDD [13]. She summarized the results of the findings as follows:

1. Weak evidence of better external quality with TDD
2. Very little evidence of better internal quality with TDD
3. Moderate evidence of decreased productivity with TDD

Summarizing many of the concerns with scaling TDD to significant projects, John McGregor states, "Design coordinates interacting entities. Choosing test cases that will adequately fill this role is difficult though not impossible. It is true that a test case is an unambiguous requirement, but is it the correct requirement?" He goes on to suggest that unlike more robust design techniques, it is not clear how TDD ensures "the correctness, completeness, and consistency of the design [14]."

So the quantitative studies of TDD are decidedly ambivalent. And the qualitative discussion argues strongly that both TDD and BDD, when applied properly to a narrow band of projects and teams, provide quality gains in terms of unambiguity, completeness, and (partially) consistency. Note that since neither process model mitigates the potential for knowledge-transfer errors, correctness is unaffected.

V. FORMAL SPECIFICATION-DRIVEN DEVELOPMENT

The objective of this paper is not merely to argue that formalism is good, but rather to demonstrate how a specific application of formalism within the software development process, together with adequate tool support, can yield the benefits of TDD and further make significant gains in both correctness and consistency attributes.

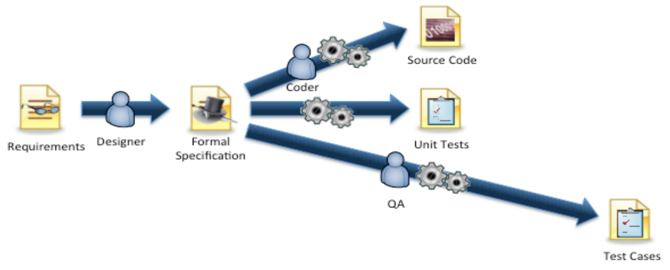

Figure 2. FSDD Artifact Production

A. *Description*

Unlike TDD and BDD, Formal Specification-Driven Development (FSDD) does not affect the underlying process model. Rather, it modifies the principle software design artifact to change the type and nature of knowledge transfer to follow. The design produces a formal specification in an organizationally selected behavioral specification language (JML, Dafny, Spec#, etc.), which becomes the artifact of record for both implementation and test agents. The introduction of a formal specification is a key step towards making significant progress improving both correctness and consistency properties. Although no additional steps have been inserted into the traditional SDLC, a new artifact has been introduced and it will likely require more rigor than prior artifacts. One may reasonably conclude that the design effort is likely to increase. However, the following sections will present a qualitative case that the additional effort would be more than offset by a reduction in re-work due to knowledge transfer errors and downstream activity efficiencies.

The artifacts produced by FSDD are illustrated in Figure 2. Arrows indicate FSDD processes and are labeled to indicate manual and automated steps. Arrows also indicate the artifact flow for input and output of each process. Automation support for artifact production reduces the risk of introducing knowledge transfer errors. The production of a formal design specification allows tool chain support in the development of source code, unit tests, and test cases.

B. Tool Requirements

Tool support is a critical component of FSDD. Even though FSDD can be manually implemented as an intellectual exercise, employing FSDD without automation is impractical at best. The following tools are anticipated in order to apply FSDD in non-trivial circumstances.

1. Design Specifiation Analyzer

The designer uses the design specification analyzer to verify the consistency and completeness of a formal design specification. Referenced entities must be defined and consistent in the specification language typing system. Although this tool does not directly reduce knowledge transfer errors, it does support quality assurance of the design phase artifacts.

2. Implementation-Stub Generator

A subset of the behavioral specification will include an interface specification. This includes such information as classes, methods, pure functions, arguments, and return types. This tool realizes behavior-less stubs in the implementation language of these entities and maps specification fundamental types (Integer) into implementation types (int). The intent of the implementation-stub generator is not behavioral code generation. It saves the coder considerable typing, and more importantly, insures a minimal, 'lexical-level' compliance with the specification. The specification must include all public access to a class, although the coder is free to elaborate private entities.

3. Specification-Aware Compiler

An FSDD-enabled compiler is expected to accept the specification as input in addition to programmer-generated source code. The compiler ensures that the source code is 'lexically' compliant to the specification as generated by the stub generator. Non-compliance will be flagged as an error. The addition of public entities not contained in the specification will also be flagged as an error. The source code will be statically analyzed and predicates that can be proven will be discharged. Unproven predicates will be injected as program logic in the target program to be validated dynamically during program execution.

4. Unit-Test Framework

The unit-test framework will generate automated unit tests from the formal specification. This tool provides the 'driven' in FSDD. The goal of the programmer is to elaborate the generated stubs with source code to create the behavior that successfully completes the unit tests and thus is compliant with the formal specification. The combination of tools and activities works together to enhance correctness.

5. Test Case Analyzer

The test case analysis tool is an aide to Quality Assurance (QA) engineers. It scans the formal specification and reports detected boundary values for subsequent use in acceptance testing against the requirements specification.

C. Issues

1. Formal Specifications Are Hard To Write

FSDD requires the software designer to learn to read and write a new, more abstract language. Additionally, designers with an implementation background will have to adopt a new approach. They must learn to think in terms of 'what' an entity does, rather than 'how' an entity does it. The requirement for this new skill is restricted to software designers. No other labor category within the process must be able to write a formal specification and designers should account for a small percentage of the complete development team.

2. Formal Specifications Are Hard To Read

Formal specification is a challenging subject. Many (perhaps most) programmers lack the training and experience to read and comprehend a formal design specification. In order to evaluate the worst-case ramifications, consider the case in which the entire coding team is unable to read the formal design specification. The specification is a sealed, black box artifact that is only comprehended by the support tools. The coding team can still use the implementation-stub generator to create the initial source files. They would still use designer-specified unit tests to validate their work. But, they would have to rely on the design support artifacts such as UML diagrams and narrative description to convey behavior. This situation does admit the potential for a knowledge transfer error. But the error is contained since coders do not write the unit tests. In those instances in which code completed behavior does not pass unit testing, the coder must coordinate with the designer to resolve the issue.

Although the above scenario is less than ideal due to the additional coordination, the outcome is still superior to the non-FSDD outcome. Without the formal specification, the knowledge transfer error is likely to remain undetected until this particular coding element is integrated with the rest of the developing system. Ideally, the coders can read and comprehend the specification. Then they need coordinate with the designer only when the specification or design documentation is in error.

3. FSDD, Validation, and Non-Functional Requirements

Formal methods and automation can assist primarily with verification. Is the artifact built the right way? Validation is quite a bit more difficult. Is this the right artifact? Answering these questions generally requires more than formal logic. Specifically, they require a qualitative judgment from a knowledgeable subject matter expert. Also, verification of non-functional requirements is challenged by the initial selection of specification language. Naturally, a behavioral specification language is specifically designed to express **behavior**, not performance characteristics. Behavioral specifications are selected as the initial focus of FSDD due to their scope at addressing knowledge transfer issues. But FSDD is not behavioral specific. The specification language can be augmented with additional logics such as temporal logic to specify performance, which is beyond the scope of this paper.

D. Advantages

The utilization of FSDD with full tool support provides numerous advantages to the development process. Similar to TDD, the goal of the programmer is to satisfy the automated tests. However, the tests are not generated by the programmer, but rather are generated by automation from the specification. The programmer continues to add implementation until full compliance is achieved as validated by both static and dynamic analysis. The module/class is then ready for integration.

1. Static Analysis: Implementation

Using the techniques reviewed above, a specification aware compiler will warn the programmer when the implementation does not satisfy its specification.

The compiler will first statically analyze the implementation for compliancy to its specification with three possible outcomes: proven non-compliant, indeterminate, and proven compliant. If non-compliant, the compiler would emit a warning message. After the programmer corrects the mistake, the compiler may be able to prove the new implementation is correct. In this case, unit testing is not required.

2. Specification Derived Unit Tests

The creation of unit tests from the specification is straightforward. The pre-conditions specify the input domain of the specification element. Analysis can readily provide test case data for input partitioning and boundary testing [8]. With test cases selected, unit testing consists of evoking the element with each case and checking the post-conditions and invariants. Any failure constitutes a failure of the unit test.

3. Static Analysis: Test

The test case analyzer can also provide material benefit to functional testing in the Quality Assurance (QA) phase of software development. With a formal behavioral specification, tooling can extract key boundary values to use in functional, black box testing without requiring input domain partitioning and boundary testing [8].

4. Dynamic Analysis: Test

After development of the test cases, integration and system testing will then be performed with instrumented builds of the implementation. Remember that an instrumented build is one in which the unproven pre-conditions, post-conditions, and invariants are asserted (validated dynamically). This approach will detect failures that might otherwise go undetected, and establish responsibility down to the lowest specified element. Kosmatov noted that software testing comprises about 50% of the total cost of software development [15]. In FSDD, the model is provided to the QA engineer without additional effort.

5. Resource Optimization (Personnel)

TDD presumes a uniformly high level of capability amongst the design team [16]. This follows from the assumption that programmers develop their own tests before coding any implementation. Since the test is the specification, each programmer shares equal responsibility for design. This process does not scale beyond small (1 – 5) developer teams for two principle reasons. First is the difficulty with attracting and keeping a large group of top-tier software developers. The second reason is fiscal justification. TDD requires uniform capability amongst the development team and would require more highly skilled software engineers than would FSDD, which does not require uniformity and shifts complex tasks onto a smaller design team resulting in reduced costs.

6. Specification Compliance Verification

During the development process, a significant percentage of total effort is expended on verification activities. According to Tian, “Software verification activities check the conformance of a software system to its specifications [8]." FSDD enforces conformity to the specification throughout the code development phase and provides designers assurance that the produced source code adheres to their intent. However, the FSDD automation cannot replace all review activities. Manual checks of new or modified code are still required to verify compliance to organizational standards.

E. Summary of Impact on Quality Attributes

Table 1 provides a summary of the results of the qualitative analysis and discussion of the process methodologies with regards to the attributes of unambiguity, completeness, consistency and correctness.

VI. PROCESS EXAMPLE

This section will provide a simplified hypothetical walkthrough of an application of Formal Specification-Driven Design (FSDD). The example application will be a simple element of a banking application, the realization of a bank account. The walkthrough will begin with its specification, continue through its implementation, and conclude with some aspects of functional testing.

Table 1: Attribute Impact Summary

Methodology	**Impact**			
	Unambig	***Complete***	***Consistent***	***Correct***
Traditional	Baseline			
TDD	Improved	Improved	Minor	Improved
BDD	Improved	Improved	Minor	Improved
FSDD	Improved	Improved	Improved	Improved

After reviewing the Software Requirements Specification (SRS), the designer completes a design specification in an organizationally appropriate specification language. The language used in this walkthrough does not represent any specific language, but was chosen for maximum simplicity and comprehension. The specification for this design is provided below.

```
class BankAcount
{ // Model Variables
  constant Decimal MAX_BALANCE = 999999999.99;
  constant Decimal MAX_TRANSACT = 99999999.99;

  // Class Predicates
  pre-condition: GetBalance() == 0;
  pre-condition: IsLocked() == false;
  post-condition: GetBalance() == 0;
  post-condition: IsLocked() == false;
  invariant: GetBalance() in [0 .. MAX_BALANCE]; …}
```

The above specification fragment declares one model variable. It exists only in the model and is used to specify behavior. The **constant** keyword declares the model variable to be un-modifiable. **Decimal** and **Boolean** are specification language fundamental data types. For each implementation language, these must be mapped to implementation data types. In this example, classes can have three differing predicates: pre-conditions, post-conditions, and invariants. The pre-condition for a class must evaluate to true at the conclusion of object construction. The post-condition for a class must evaluate to true before object destruction. The invariant for a class must evaluate to true after class construction, before and after any specification methods, and before object destruction. Class method specifications continue below.

```
// Class Methods
void Credit(Decimal[0 .. MAX_TRANSACT] amount)
{ pre-condition: not IsLocked();
  post-condition: post GetBalance() == pre GetBalance()
  + amount;}
```

```
void Debit(Decimal[0 .. MAX_TRANSACT] amount)
{ pre-condition: not IsLocked();
  post-condition: post GetBalance() == pre GetBalance()
  - amount;}

void Lock()
{ pre-condition: not IsLocked();
  post-condition: IsLocked();}

void Unlock()
{ pre-condition: IsLocked();
  post-condition: NOT IsLocked();}

pure Boolean IsLocked();
pure Decimal GetBalance();
```

Each of these methods must exist in a compliant implementation. Additionally, the two **pure** methods, IsLocked() and GetBalance(), must have no side effects. This enhances the ability of the static analyzer to reason about them. After examination with the design specification analyzer, this design specification is committed to source version control. At some later point, the complete (or partially-complete) design is delivered for coding. In this example, the system is to be built in C++. The programmer assigned to implement the BankAccount class begins with the stub generator, which creates two files, BankAccount.h and empty methods with signatures in BankAccount.cpp:

BankAccount.h:

```
class BankAccount
{public:
 const BCD_Type MAX_BALANCE(999999999.99);
 const BCD_Type MAX_TRANSACT(99999999.99);
 void Credit(BCD_Type amount);
 void Debit(BCD_Type amount);
 void Lock();
 void Unlock();
 bool IsLocked() const;
 BCD_Type GetBalance() const;};
```

If coders were to modify any method names, parameters, or attributes they will receive an error message from the compiler, because the source is no longer compliant with its specification. The coder compiles the source file and is rewarded with compiler error messages. The formal methods have not come into play yet. Any traditional C++ compiler would fault the same conditions.

The programmer would then make the modifications to fix the compiler errors to include missing return types. At this point, the FSDD-enabled compiler is able to improve the standard process. Since the compiler has access to the expected behavior, compiling this code would now yield a series of "missing functional implementation" warning messages for each of the empty methods.

The workflow proceeds much as it would under TDD/BDD. The object is to make the errors/warnings go away. So they start to implement the missing behavior. They add behavior, get compiler messages, each time fleshing out more of the code. The static analysis phase of the compiler knows from class pre-conditions that a specific method should return a value after construction. But, if it is missing a functional implementation, the compiler is able to reason that the specified behavior is unlikely to be implemented. Similarly, post-conditions are analyzed and may be shown not to hold in all execution paths. The coder then corrects the errors and continues the implementation until the source code compiles without errors or warnings.

The coder is now ready for unit testing. The coder uses the unit-testing framework to generate the test harness and test cases. The framework examines the specification parameters, pre-conditions and invariants to partition input/class state for selection of test cases. For example, Credit() takes an input Decimal type ranging from 0 to 99999999.99. One reasonable automated partitioning might be minimum, minimum + 1, average, maximum – 1, and maximum or {0, 1, 49999999.99, 99999998.99, 99999999.99}. The framework's test harness would invoke Credit() with each of these parameters and verify post-conditions and invariants. The process is repeated for every specified entity in the source file. If the unit testing reports an error in any method, the coder will again correct the code until the implementation of BankAccount compiles without warnings or errors and all unit tests pass. This signals task completion. Both the coder and the designer have initial confidence that the implementation complies with the design specification. A senior programmer will still need to perform a code review to validate that the implementation uses appropriate data structures, employs suitable algorithms, and adheres to coding standards.

The BankAccount module is now ready for integration by Quality Assurance (QA) engineers. They use the test case analyzer to reveal input partitioning values much as with unit testing above. It aids them in the creation of the integration test cases, which are then run against an instrumented build of the system to dynamically verify specification predicates. A manually created rendition of an instrumented BankAccount follows. Instrumentation code has been highlighted in yellow. If a predicate within an assert() evaluates as false, then a runtime exception is thrown which halts the application and displays identifying information. The instrumented build would be used for integration and system testing. If the performance of the instrumented system was acceptable for deployment, then it could also be used in acceptance testing and delivered as the final product. Otherwise, instrumentation would have to be removed prior to performance and acceptance testing. Also note that static analysis could absolve the need for some of the instrumentation. If an invariant or method post-condition could be proven to hold, then those predicates would not need to be asserted. Similarly, if a method pre-condition could be proven to hold at all call-sites, then its assertion could also be removed.

```
BankAccount::BankAccount()
{ balance = BCD_Type(0);
  locked = false;
  // class pre-conditions
  assert(GetBalance() == 0);
  assert(IsLocked() == false);
  // pre-condition 1 => invariant}

BankAccount::~BankAccount()
{ // post-condition 1 => invariant
  // class post-conditions
  assert(GetBalance() == 0);
  assert(IsLocked() == false);}

void BankAccount::Credit(BCD_Type amount)
{ // pre-conditions
  assert(amount >= 0) && (amount <= MAX_TRANSACT));
  assert(IsLocked() == false);
  // invariant
  assert((GetBalance() >= 0) &&
     (GetBalance() <= MAX_BALANCE));
  // post-conditions require two-state
  BCD_Type preBalance = GetBalance();
  balance.Add(amount);
  // invariant
  assert((GetBalance() >= 0) &&
     (GetBalance() <= MAX_BALANCE));
  // post-conditions
  assert(GetBalance() == preBalance + amount);}

void BankAccount::Debit(BCD_Type amount)
{ // pre-conditions
  assert(amount >= 0) && (amount <= MAX_TRANSACT));
  assert(IsLocked() == false);
  // invariant
  assert((GetBalance() >= 0) &&
     (GetBalance() <= MAX_BALANCE));
  // post-conditions require two-state
  BCD_Type preBalance = GetBalance();
  balance.Subtract(amount);
  // invariant
  assert((GetBalance() >= 0) &&
     (GetBalance() <= MAX_BALANCE));
  // post-conditions
  assert(GetBalance() == preBalance - amount);}

void BankAccount::Lock()
{ // pre-conditions
  assert(IsLocked() == false);
  // invariant
  assert((GetBalance() >= 0) &&
     (GetBalance() <= MAX_BALANCE));
  locked = true;
  // invariant
  assert((GetBalance() >= 0) &&
     (GetBalance() <= MAX_BALANCE));
  // post-conditions
  assert(IsLocked() == true);}

void BankAccount::Unlock()
{ // pre-conditions
  assert(IsLocked() == true);
  // invariant
  assert((GetBalance() >= 0) &&
     (GetBalance() <= MAX_BALANCE));
  locked = false;
  // invariant
  assert((GetBalance() >= 0) &&
     (GetBalance() <= MAX_BALANCE));
  // post-conditions
  assert(IsLocked() == false);}

bool BankAccount::IsLocked() const
{ // invariant
  assert((GetBalance() >= 0) &&
     (GetBalance() <= MAX_BALANCE));
  return locked;
  // invariant: not required on exit from pure method}

BCD_Type BankAccount::GetBalance() const
{ // invariant
  assert((GetBalance() >= 0) &&
     (GetBalance() <= MAX_BALANCE) );
  return balance;
  // invariant: not required on exit from pure method
```

VI. CONCLUSIONS AND FUTURE WORK

As consumers, we routinely and knowingly purchase defective software. We proceed to install it on our computers and entrust our sensitive and valuable data to it. We don't really trust the system manipulating our data, so we back it up. Sometimes, we also don't trust the backup system and back it up as well. When colleagues lose important work due to a software application freezing or becoming unresponsive, we advise them to save more often. Not only do we accept that software is defective, we expect it. As users, we demand new features when the existing ones only mostly work. Corporations staff entire departments to be on call and assist employees when software fails. If architects designed with equivalent defect densities, buildings would only mostly stay up, people would only enter when necessary, would never wander far from an exit, and would keep emergency services on speed dial while inside.

Reliably producing quality software requires discipline and rigor. And verification technologies are not a panacea for all software development challenges. But formal specifications and FSDD mitigate many of the current difficulties. Future work in this area extends from basic research to tool implementation. In order to gain industry traction, a single, general-purpose behavioral specification language needs to emerge with the needed language features. There is still much work to be done in static analysis. Also, the efficiency of current static analysis techniques needs to improve sufficiently to allow completion during an ordinary compilation operation. Current strategies require much more time than compilation itself. Existing tools must also be adapted to fit the descriptions outlined in the previous section. In order to take advantage of modern, multi-core systems, behavioral specification languages must be extended to allow reasoning about concurrency. Finally, in order to spur widespread adoption by the software industry, objective, empirical, quantitative studies of FSDD and formal specification in general must be conducted to establish its business value.

RERERENCES

[1] K. Beck, *Test-Driven Development: By Example*. Boston: Addison-Wesley, 2003.

[2] M. Wynne, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Dallas, Tex: Pragmatic Bookshelf, 2012.

[3] F. Tsui, "Process: Definition and Communication.," in *Encyclopedia of Software Engineering*, 2010, pp. 715–728.

[4] "The Verified Software Initiative," 2008. [Online]. Available: http://qpq.csl.sri.com/vsr/vsi.pdf/view. [Accessed: 14-Feb-2013].

[5] Chelf, Ben, "Measuring Software Quality: A Study of Open Source Software," Coverity, Inc., 2006.

[6] C. A. R. Hoare, J. Misra, G. T. Leavens, and N. Shankar, "The Verified Software Initiative: A Manifesto," *ACM Comput Surv*, vol. 41, no. 4, pp. 22:1–22:8, Oct. 2009.

[7] B. Meyer, *Object-Oriented Software Construction (2nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.

[8] J. Tian, *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement*. Hoboken, N.J: Wiley, 2005.

[9] S. L. Pfleeger, *Software Engineering: Theory and Practice*, 4th ed. Upper Saddle River [N.J.]: Prentice Hall, 2010.

[10] K. Beck, *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley, 2000.

[11] H. Erdogmus, M. Morisio, and M. Torchiano, "On the Effectiveness of the Test-First Approach to Programming," *Softw. Eng. IEEE Trans. On*, vol. 31, no. 3, pp. 226–237, 2005.

[12] "Introducing BDD," *Dan North & Associates*. .

[13] S. Kollanus, "Test-Driven Development - Still a Promising Approach?," in *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, 2010, pp. 403–408.

[14] S. Fraser, D. Astels, K. Beck, B. Boehm, J. McGregor, J. Newkirk, and C. Poole, "Discipline and Practices of TDD: (Test Driven Development)," in *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, New York, NY, USA, 2003, pp. 268–270.

[15] N. Kosmatov, "Constraint-Based Techniques for Software Testing," in *Artificial Intelligence Applications for Improved Software Engineering Development*, F. Meziane and S. Vadera, Eds. IGI Global, 2009.

[16] Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.