On the Verification and Validation of Software Modules: Applications in Teaching and Practice

F. Tchier
King Saud University
Riyadh, Saudi Arabia

A. Mili
NJIT, University Heights
Newark, NJ 07102-1982

Abstract

Modular programming is a discipline of bottom-up programming that is based on the principle of information hiding, whereby each module of a software product exports its specification but hides its design. This principle offers many advantages, including support for better maintainability, better testability, better reliability, and better reusability. Object Oriented programming languages support modularity, but they do not substitute for a sound discipline of modular programming. In this paper, we present a formal discipline of modular programming, report on our experience using it in undergraduate and graduate courses, and speculate on its potential for scaling up.

1. Introduction: A Discipline of Programming

Modular programming [Parnas, 1972] is a discipline of bottom-up programming that is based on the principle of information hiding, whereby each module in a software product exports its specification but hides its design and implementation. This discipline offers significant advantages to software developers, in terms of:

- Better maintainability: Thanks to information hiding, modules are designed and implemented independently, thereby precluding that a fault in the design of one module causes a fault in another.
- Better testability: Since each module carries its own specification, it is easier to test locally, and to distinguish between the failure of individual modules and the failure of module interactions.
- Better reliability: The specification of a module serves as a contract between the developer of the module and its user, thereby reducing the likelihood of faults arising from miscommunication between developers.
- Better reusability: it is widely recognized that modular programming, being a bottom-up design discipline, provides better support for software reuse than top down design.
Object oriented languages offer some support for modular programming through their focus on objects and classes, their separation of specification from implementation, their support for software reuse, and more broadly through their compatibility with bottom up software design. But using an Object Oriented language is hardly a substitute for adhering to a sound discipline of modular programming: There is far more to the latter than the former, and using an OO language is neither a necessary condition nor a sufficient condition to adhering to a discipline of modular programming.

If we consider, for example, the distinction between the specification and the implementation of a module, object oriented languages enable us to invoke a module on the basis of its specification, and to detail its implementation in a separate file/ unit. But their representation of specifications is at odds with elementary principles of good specification, such as formality and abstraction. For example, module specifications are typically represented by a list of method signatures, possibly documented in natural language ---hardly what we would call a formal specification. Also, if we complement this description by a model-based formal specification such as Z [Davies and Woodcock, 1996] or B [Lano and Houghton, 1996], then we hardly leave any latitude to the programmer, since then the specification is dictating a model of the module’s data, as well as an individual specification of each method of the module. By contrast, we propose in this paper a formal specification notation that focuses exclusively on externally observable module behavior, and imposes no constraints on how data is structured, nor on how methods modify the data.

We have used this discipline of specification and verification in the context of two separate courses: an undergraduate course on data structures; and a graduate course on software specification, verification, and testing. These courses will be discussed in section 2. Beyond their use in the classroom, we believe that this discipline can be used in practice in the specification, design and verification of ADT’s, or OOP modules in general; our premise is based on classroom observations, to the effect that students find the approach intuitively appealing, and that they practice it with great ease within the confines of a fifteen-week term. In sections 3, 4, and 5, we discuss in turn, how we propose to specify, verify and test modules using our approach; in section 6, we discuss how to estimate the reliability of a software product from an empirical analysis of its performance under test. Finally, in section 6 we summarize our findings and our conclusions, and we sketch directions of future research.

2. Teaching Practice

We are using our discipline of programming in two CS courses, an undergraduate course on data structures, and a graduate course on verification and validation. We briefly review these in turn, below:

2.1 An undergraduate course on data structures

In a programming curriculum, we view the data structures course as an opportunity to acquaint the student with modular programming; a key idea of modular programming is information hiding [Parnas, 1972], which is embodied by the sharp distinction between the specification of a module and its implementation. Many data structures textbooks, even when they purport to uphold software engineering principles, are not rigorous about this distinction, and often specify the data type using implementation details, let implementation considerations affect the definition of the data type, leave much ambiguity in
the definition of the data type, etc. In a recent textbook on data structures [Tchier and Mili, 2010], we have adopted the following three-step discipline for discussing data structures:

- First, we introduce the data type by means of behavioral formal specifications, that describe the functional attributes of the data type, but give no insight into how the data type may be implemented; specifications are represented by axioms and rules, where axioms reflect the behavior of the data type for simple data values and rules define their behavior inductively for more complex data values. These axiomatic specifications are validated for completeness against independently generated validation data.

- In the second phase, we discuss the usage of the data type, based on its validated specification. The issue we explore with the students is: given a data type that satisfies these specifications, what are possible uses/applications of it? As we explore applications of the data type, we provide the students with hidden implementations of the data type, and impress upon them, concretely, that they can use these data types without knowing any detail about their implementation ---all they need to know is their specification.

- In the third phase, we switch sides, from the party using the data type to the party implementing it. We explore various candidate implementations of the data type, and revisit the applications discussed in phase 2, where we replace the hidden implementations used in phase 2 by the implementations developed by students, and ensure that the applications behave in the same manner.

Whereas people generally think of the contrast between ACM’s CS1 and CS2 courses in terms of the distinction between algorithms and data structures, we think of it in terms of the distinction between structured (top down) programming and modular (bottom up) programming. We believe that the data structures course ought to be used as an opportunity to teach students about the tenets of modular programming, and the important principles inherent therein.

2.2 A Graduate Course on Verification and Validation

NJIT’s Master of Software Engineering Program includes among its core courses, a course dealing with verification and validation. In this course, students proceed through a term project conducted by teams of four students, structured as follows:

- A specification phase, when students select a complex ADT and specify it using the axiomatic specification alluded to above (and discussed in detail in section 3); sample ADT’s are selected to be sufficiently complex that their implementation is expected to take 300 to 600 lines of code.

- A validation phase: Each team is actually divided between a specification generation group (three students, typically) and a specification validation group (one student). Also, the specification phase is divided between two steps:
  - The specification step, when the generation group produces the axiomatic specification while the validation group produces validation data that the specification is supposed to comply with.
  - The validation step, when the validation group confronts the generated specification against the validation data and reports discrepancies back to the generation group for revision, clarification, etc.
For the sake of redundancy, these two groups do not communicate with each other during the generation phase, and work independently from a common source (typically: The list of methods of the ADT, along with a summary informal description of each method). The outcome of the specification phase is a validated axiomatic specification.

<table>
<thead>
<tr>
<th>Team Step</th>
<th>Specification team (three members)</th>
<th>Validation team (one member)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification step (three weeks)</td>
<td>Generating the Axiomatic Specification From the ADT’s informal description</td>
<td>Generating Validation Data from the ADT’s informal description</td>
</tr>
<tr>
<td>Validation step (one week)</td>
<td>Updating the specification/ providing rationale/ Reviewing interpretation</td>
<td>Validating the Specification against generated validation data/ Negotiating with the specification team</td>
</tr>
</tbody>
</table>

• **An implementation phase:** During the implementation phase, the whole team regroups again and develops an implementation of the ADT in a language of their choice, subject to two constraints: first to choose the simplest possible implementation and invoke the simplest programming constructs of the language, as they have to prove the correctness of their code subsequently; second, to abstain from executing the code or testing it. They are allowed to compile the code, but not to execute it nor debug it.

• **A verification phase:** In the verification phase, students are expected to use Hoare logic [Hoare, 1969] to verify that their code satisfies the axioms of the specification. Because axioms typically specify the behavior of the ADT for base data, they usually invoke simple fragments of code, and are usually easy to prove by Hoare logic; in particular, because they deal with base data, they do not involve burdensome task of generating invariant assertions or complex intermediate assertions.

• **A testing phase:** While we use axioms as specifications against which we prove the correctness of the code by means of Hoare’s logic, we use rules of the specification as oracles against which we test the implementation. Specifically, we develop test drivers for the ADT, involving automatic generation of random test data according to selected usage patterns, and we check at run-time that the properties of the rules are satisfied for an arbitrarily large set of data configurations. Paradoxically, the purpose of this phase is not to remove faults in the ADT implementation, but rather to ensure that the test driver (generating random test data, invoking the ADT, checking that rules are satisfied, etc) is correct, or at least that we remove the most salient faults from it.

• **A reliability estimation phase:** In the reliability estimation phase, we use the test driver developed in the previous phase to locate and remove faults from the ADT. Specifically, we adopt the Cleanroom testing strategy [Mills et al, 1987], which uses randomly generated test data according to a predefined usage pattern to test the ADT, discover its faults and remove them. In the process, we monitor the evolution of the reliability of the ADT as faults are removed, and estimate the resulting reliability when testing is concluded. We adopt a simplified version of the Cleanroom reliability model, which provides the estimated reliability as follows:
\[ MTTF_n = MTTF_0 \times R^N, \]

where \( MTTF_n \) is the estimated reliability after the removal of \( N \) faults, and \( R \) (hopefully greater than 1) is the reliability growth factor, i.e. the factor by which the estimated reliability of the product grows after each fault is removed.

We believe that this approach, while being reasonably tractable, exposes the graduate student to a number of important software engineering principles, such as: the importance of software specifications; the practice of lightweight formal specifications; the meaning and the importance of specification validation; the meaning and importance of program verification; the discipline of goal oriented, oracle-based testing; the practice of functional testing; the meaning of usage pattern, and its use in test data generation; the meaning of reliability modeling; the practice of making quantifiable, verifiable claims, and supporting them with empirical evidence.

3. A Discipline of Specification

3.1. A Specification Model

There is ample evidence in the literature [Frappier and Habrias, 2006] for the importance of carefully specifying software requirements prior to software product development. The specification phase of the software lifecycle has always played a central role in software engineering, not only because of its intrinsic technical interest, but also because it is critical in determining the success or failure of software projects. Our challenge in adopting a specification discipline is to satisfy a wide range of conflicting criteria [Meyer, 1985]:

- **Simplicity.** The notation must be simple, so that software engineers have no trouble adopting it, and users have no trouble understanding it.
- **Formality.** The notation must be formal, so as to make it possible to validate the specification for completeness and minimality, to provide automated support for specification generation and validation, to use the specification as an oracle in testing, to use the specification for product verification, etc.
- **Abstraction.** The specification notation must make it possible to represent the desired functional attributes of the software product that we are specifying without dictating its design attributes or design constraints. It is by virtue of this criterion that we rule out model-based specifications, since these specify the software product by building a candidate model for it, all be it an abstract model.

In keeping with these criteria, we propose to represent the specification of modules (such as abstract data types, object oriented classes, objects, etc) by means of the following artifacts:

- **An input space, \( X \),** which is defined as the set of input signals that a module may accept; in programming terms, this is the set of method names that are visible/accessible from outside the
module, along with their parameters, if any. For the sake of argument, we distinguish between two types of input signals:

- **V-operations**, whose invocation is expected to yield a value; we usually denote it with $X_V$.
- **O-operations**, whose invocation does not produce any value, but may affect future behavior of the module; we usually denote it with $X_O$.

From the input space, $X$, we construct the set of input histories, which is made up of all the sequences of input signals; we usually denote it with $H$.

- **An output space**, $Y$, which is defined as the set of output values that the module may return following the invocation of a V-operation.
- **A relation from $H$ to $Y$**, which we usually denote by the name of the module. This relation maps input histories to outputs; for the sake of simplicity, we assume this relation to be deterministic, even though this is not always the case.

As an illustrative example, we consider a stack of items of type itemtype. We present in turn its input space, its output space, and its relation.

- **Input space**, $X = \{\text{init, pop, top, size, empty}\} \cup \{\text{push}\} \times \text{itemtype}$.  
  - $X_V = \{\text{top, size, empty}\}$.  
  - Operation **top** returns the top of the stack (last element stored) if the stack is not empty; else it returns an error message. Operation **size** returns the number of elements of the stack. Operation **empty** returns true if and only if the stack is empty.
  - $X_O = \{\text{init, pop, push}\}$.  
  - Operation **init** (re) initializes the specification to its empty status, erasing all past history. Operation **pop** removes the top (most recently pushed) element of the stack, if the stack is not empty; else it leaves the stack unchanged. Operation **push** pushes an element (provided as parameter) on top of the stack.

From the set of input symbols $X$, we derive the set of input histories, $H=X^*$.

- **Output space**, $Y = \{\text{error}\} \cup \text{itemtype} \cup \text{integer} \cup \text{boolean}$.  
  The output space is the union of four terms: the **error** message arises whenever an illegal input history is submitted; a result of type **itemtype** is returned whenever a top operation is submitted and the stack is not empty; a result of type **integer** is returned whenever operation **size** is submitted; and finally, a result of type **boolean** is returned whenever operation **empty** is submitted.

- **Relation from $H$ to $Y$**: We let stack be the relation from $H$ to $Y$ that, to each input history $h$ associates the output that we want candidate implementations to return. We give below sample input/output pairs of this relation.
  - stack(init.top.push(a).push(b).push(c).size)=3.
  - stack(init.pop.push(a).pop.push(b).pop.empty)=true.
  - stack(pop.init.pop.push(a).top.empty.empty)=false.
Note how this notation pursues the goal of abstraction: we are specifying the stack ADT without giving any detail pertaining to structure/design/implementation.

3.2. Axiomatic Specifications

For all its simplicity, formality, and abstraction, our specification notation is not useful in practice unless we find a way to represent the relation of the specification in a closed form (representing an infinite relation with a finite/compact representation). We propose an axiomatic representation, where

- Axioms represent the behavior of the module for trivial input histories, and
- Rules relate the behavior of the module for complex input histories to its behavior for simpler input histories.

We illustrate this axiomatic notation by the following specification of the stack ADT.

**Axioms for the stack:**

1. **Top axioms.**
   a. \( \text{stack(init.top)} = \text{error}. \)
   b. \( \text{stack(init.h.push(a).top)} = a. \)

2. **Size axiom.**
   \( \text{stack(init.size)} = 0. \)

3. **Empty axioms.**
   a. \( \text{stack(init.empty)} = \text{true}. \)
   b. \( \text{stack(init.push(a).empty)} = \text{false}. \)

**Rules for the stack:**

Let \( h, h' \) be arbitrary input histories, and \( h+ \) be a non empty input history.

1. **Init rule:**
   \( \text{stack(h.init.h')} = \text{stack(init.h')}. \)
   \text{Init} reinitializes the state of the stack: whether it received history \( h \) prior to \text{init} or not makes no difference now (\( h'=() \)) nor in the future (\( h'\neq() \)).

2. **Init Pop rule:**
   \( \text{stack(init.pop.h)} = \text{stack(init.h)}. \)
   \text{Pop} on an empty stack has no impact now (\( h=() \)) nor in the future (\( h\neq() \)).

3. **Push pop rule:**
   \( \text{stack(init.h.push(a).pop.h+)} = \text{stack(init.h.h+)}. \)
A pop operation cancels the push that precedes it: whether we push a then pop it or do neither, makes no difference in the future (h+≠()).

4. Size rule:
   stack(init.h.push(a).size)=1+stack(init.h.size).
   Each push operation necessarily increases the size of the stack by 1, because the stack size is not bounded.

5. Empty rules
   a. stack(init.h.push(a).h’.empty) ⇒ stack(init.h.h’.empty).
      If, despite having operation push(a) in its history, the stack is empty, then a fortiori it would empty without push(a).
   b. stack(init.h.empty) ⇒ stack(init.h.pop.empty).
      If the stack is empty, then a fortiori it would be empty if an extra pop operation was performed in its past history.

6. V-operation rules
   a. stack(init.h.top.h+)=stack(init.h.h+).
   b. stack(init.h.size.h+)=stack(init.h.h+).
   c. stack(init.h.empty.h+)=stack(init.h.h+).
   V-operations have no impact on the future behavior of the stack, by definition, since all they do is to enquire about its state.

Using this axiomatic system, we can prove (or, making the closed world assumption) disprove any statement of the form: stack(h)=y, where h is an arbitrary history and y is an output symbol; this formula says that execution of the sequence h yields the output y.

3.3. Specification Validation

Given the importance of the specification phase, and the criticality of making sure that specifications are valid before we proceed through subsequent phases of the lifecycle, we ought to ask the question [Sommerville, 2010]: how do we ensure that specifications are valid, and reflect all the relevant requirements and constraints? The purpose of the specification validation phase is to answer this question.

In section 2.2, we had discussed a two-phase, two-activity lifecycle for the requirements specification phase of the overall software lifecycle. The validation activity of this lifecycle proceeds as follows:
In the specification generation phase, the specification validation activity produces independent validation data that the specification being generated must comply with. We propose that this data take the form of \((h,y)\) pairs, where \(h\) is an input history and \(y\) is the output that the validation team believe should be produced for the input history \(h\).

In the specification validation phase, the specification validation activity checks whether the generated specification does indeed comply with the independent validation data. We propose that this be done by checking whether the formula \(M(h)=y\) is a theorem in the deductive system defined by the axioms and rules of the specification (where \(M\) is the name of the relation of the specification); in other words, this amounts to checking whether the formula \(M(h)=y\) can be inferred from the axioms and rules of the specification.

As an illustration, we consider the following validation data, which we assume was independently generated by the validation team to validate the stack specification:

- \(\text{stack}(\text{pop.init.top.pop.push(3).size.push(1).top.pop.push(5).top.pop.size.top})=3\)
- \(\text{stack}(\text{push(2).init.init.pop.push(3).top.size.push(2).push(5).top.pop.push(3).size})=3\)
- \(\text{stack}(\text{pop.init.pop.pop.push(1).size.empty.push(1).pop.top.push(1).push(1).empty})=false.\)

We review these formulas in turn, to see whether they are theorems of the deductive systems defined by our specification of the stack.

\[
\text{stack}(\text{pop.init.top.pop.push(3).size.push(1).top.pop.push(5).top.pop.size.top}) =
\]

\[
\begin{align*}
\text{init} & \text{. pop.push(3). push(1). pop.push(5). pop. top} &= \text{by virtue of the push-pop rule, applied twice} \\
\text{init. pop.push(3). top} &= \text{by virtue of the second top axiom, with h = \langle pop\rangle} \\
3 &= \text{QED}
\end{align*}
\]

\[
\text{stack}(\text{push(2).init.init.pop.push(3).top.size.push(2).push(5).top.pop.push(3).size}) =
\]

\[
\begin{align*}
\text{init} & \text{. pop.push(3). top} &= \text{by virtue of the V-op rules} \\
\text{init. pop.push(3). top} &= \text{by virtue of the init rule} \\
\text{init. pop.push(3). top} &= \text{by virtue of the V-op rules}
\end{align*}
\]
\[
\text{stack(} \text{init.pop.push(3).push(2).push(5).pop.push(3).size)} \\
= \{ \text{by virtue of the push-pop rule} \}
\]

\[
\text{stack(} \text{init.pop.push(3).push(2).push(3).size)} \\
= \{ \text{by virtue of the size rule, with } h = \langle \text{pop.push(3).push(2)} \rangle \}
\]

\[
1 + \text{stack(} \text{init.pop.push(3).push(2).size)} \\
= \{ \text{by virtue of the size rule, with } h = \langle \text{pop.push(3)} \rangle \}
\]

\[
1 + 1 + \text{stack(} \text{init.pop.size)} \\
= \{ \text{by virtue of the init-pop rule} \}
\]

\[
1 + 1 + 1 + \text{stack(} \text{init.size)} \\
= \{ \text{by virtue of size axiom} \}
\]

\[
1 + 1 + 1 + 0 \\
= \{ \text{arithmetic} \}
\]

3.

\[
\text{QED}
\]

\[
\text{stack(} \text{pop.init.pop.pop.push(1).size.empty.push(1).pop.top.push(1).push(1).empty)} \\
= \{ \text{by virtue of the init rule} \}
\]

\[
\text{stack(} \text{init.pop.pop.push(1).size.empty.push(1).pop.top.push(1).push(1).empty)} \\
= \{ \text{by virtue of the init-pop rule, applied twice} \}
\]

\[
\text{stack(} \text{init.push(1).size.empty.push(1).pop.top.push(1).push(1).empty)} \\
= \{ \text{by virtue of the V-op rules} \}
\]

\[
\text{stack(} \text{init.push(1).push(1).pop.push(1).push(1).empty)} \\
= \{ \text{by virtue of the push-pop rule} \}
\]

\[
\text{stack(} \text{init.push(1).push(1).push(1).empty)} \\
\Rightarrow \{ \text{by virtue of the empty rule, with } h=\langle \text{push(1).push(1)} \rangle, h'=\langle \rangle \}
\]
stack(init.push(1).push(1).empty)
⇒ \{by virtue of the empty rule, with h=\langle push(1) \rangle, h'=\langle \rangle \}

stack(init.push(1).empty)
⇒ \{by virtue of the second empty axiom\}
false.

From

stack(pop.init.pop.pop.push(1).size.empty.push(1).pop.top.push(1).push(1).empty) ⇒ false,

we infer

stack(pop.init.pop.pop.push(1).size.empty.push(1).pop.top.push(1).push(1).empty) = false.

\textbf{QED}

\section{Verification}
Once the specification of the module is generated and validated, we are now ready to use it to develop the product and to verify its correctness. Note that, given how the specification is written, the programmer has all the necessary latitude to implement it any way she/he wishes. We assume that it was implemented using an array and an index.

\subsection{Axiomatic Verification}
We present below a brief axiomatization of Hoare’s logic for proving programs correct with respect to specifications represented as pre/post-conditions. Formulas in Hoare’s logic take the form

\[ \{ p \} \ S \ \{ q \} \]

where \( p \) and \( q \) are assertions, and \( S \) is a programming language statement. This formula is interpreted as: If \( p \) holds prior to execution of \( S \) and \( S \) executes and terminates then \( q \) holds after the execution of \( S \). We present axioms and rules that allow us to infer complex Hoare formulas from simpler formulas.

\textbf{Axioms.} We adopt the following three axioms:

- Any tautology of logic is an axiom.
- Any formula of the form \( \{ false \} \ S \ \{ q \} \) is an axiom, for any \( S \) and any \( q \).
- Any formula of the form \( \{ p \} \ S \ \{ true \} \) is an axiom, for any \( S \) and any \( p \).
**Rules.** We present one rule for each programming language statement, in addition to a consequence rule that allows us to generalize a pre/post specification.

- **Assignment Rule:**
  \[
  q(s) \Rightarrow r(E(s)) \\
  \{q\} s = E(s)(r)
  \]

- **Sequence Rule:**
  Find some predicate \( int (\text{intermediate}) \)
  \[
  \{q\} p1\{int\} \\
  \{int\} p2\{r\} \\
  \{q\} p1; p2\{r\}
  \]

- **If-Then-Else Rule**
  \[
  \{q \land t\} P\{r\} \\
  \{q \land \neg t\} P'\{r\} \\
  \{q\} \text{if } t \text{ then } P \text{ else } P'\{r\}
  \]

- **If-Then-Rule**
  \[
  \{q \land t\} P\{r\} \\
  \{q \land \neg t\} \Rightarrow r \\
  \{q\} \text{if } t \text{ then } P\{r\}
  \]

- **While Rule**
  Find some predicate inv (invariant) and prove:
  \[
  q \Rightarrow inv \\
  \{inv \land t\} b \{inv\} \\
  inv \land \neg t \Rightarrow r \\
  \{q\} \text{while } t \text{ do } b \{r\}
  \]

- **Consequence Rule**
  \[
  p \Rightarrow p' \\
  q' \Rightarrow q \\
  \{p'\} S(q') \\
  \{p\} S(q)
  \]

4.2 Developing an Implementation

On the basis of the validated specification developed above, we propose the following stack implementation, written in (GNU) C++. In the subsequent section, we discuss how we prove this implementation with respect to the axiomatic specification given in section 3.
const int stacksize = 100;
typedef int sitemtype;
typedef int indextype;
class stack
{ public:
  stack();  // default constructor
  void init();
  // initializes or re-initializes the stack
  bool empty () const;  // tells whether stack is empty
  void push (sitemtype sitem);
  // add sitem to the stack
  void pop ();
  // deletes top of the stack
  sitemtype top ();
  // returns top of the stack
  int size ();
  // returns size of the stack
private:
  // array-based implementation.
  sitemtype sarray [stacksize];
  indextype sindex;};

#include "stack.h"
// uqueue.h header file.
stack :: stack ():
  sindex (0), {  };

stack :: init ():
{  sindex =0;};

bool stack :: empty () const
{return (sindex==0);}

void stack :: push (sitemtype sitem)
{  sindex=sindex+1; sarray[sindex]=sitem; }

void stack :: pop ()
{  if (sindex>0) {  // stack is not empty
      sindex=sindex-1;  }
}

sitemtype stack :: top ()
{  int error = -9999;
    if (sindex>0)  {return sarray[sindex];}
    else  {return error;  }
}

int stack :: size ()  {return sindex;}
4.3 Proving Implementations Against Axioms

We envision to map each axiom of the specification into a Hoare formula, which we then prove using the inference system presented above. This method is best illustrated on an example. Let us consider, for example, the following axiom:

Push Top Axiom: \( \text{stack(init.push(a).empty)} = \text{false} \),

for an arbitrary item \( a \).

To check that the implementation of the module satisfies this axiom, we propose to prove the following formula in Hoare’s inference system, where \( y \) is a variable of type boolean:

\[
\begin{align*}
&v: \{\text{true}\} \text{ init(); push(a); } y=\text{empty (); } \{y=\text{false}\} \\
&\text{The axiom provides that the sequence } \text{init.push(a).empty } \text{return false. To check whether our implementation satisfies this axiom, we invoke the sequence } \text{init(); push(a), then we call the V-op empty()} \text{ and put its result in variable } y, \text{ and check whether } y \text{ contains the value false. As for the precondition, we choose true, since init builds the stack from scratch, hence we need not know anything prior to executing init(). Now, in order to prove this formula, we must replace each method call by its body, and we replace the call of empty()} \text{ by its body, in which we replace each return statement by assignments to } y. \text{ In order to avoid confusion we represent assignment statements in C++ by := and equality by =. This yields:}
\end{align*}
\]

\[
\begin{align*}
&v: \{\text{true}\} \\
&\quad \text{sindex :=0; } \\
&\quad \text{sindex=sindex+1; sarray[sindex]=sitem;} \\
&\quad y:=(sindex=0) \\
&\quad \{y=\text{false}\}
\end{align*}
\]

In order to prove formula \( v \), we apply the sequence rule three times, and generate the following formulas:

\[
\begin{align*}
&v0: \{\text{true}\} \text{ sindex:=0 } \{\text{sindex=0}\} \\
&v1: \{\text{sindex=0}\} \text{ sindex:=sindex+1; } \{\text{sindex=1}\} \\
&v2: \{\text{sindex=1}\} \text{ sarray[sindex]:=a; } \{\text{sindex=1}\} \\
&v3: \{\text{sindex=1}\} y:=(sindex=0) \{y=\text{false}\}
\end{align*}
\]

The assignment statement rule applied to \( v0, v1, v2, v3 \) yields, respectively:

\[
\begin{align*}
&v00: \text{true } \Rightarrow 0=0, \\
&v10: \text{sindex=0 } \Rightarrow \text{sindex+1=1}, \\
&v20: \text{sindex=1 } \Rightarrow \text{sindex=1},
\end{align*}
\]
v30: sindex=1 \Rightarrow (\text{sindex}=0)=false,

all of which are tautologies, hence axioms of Hoare logic. This concludes the proof, and establishes the
validity of v. Note how trivial the proof is: by proving the implementation against the axioms
(representing the behavior of the stack on base cases) we have spared ourselves the difficulty that is
usually associated with program verification methods; of course the proof is incomplete, and must be
complemented with the testing step, which we discuss in section 5.

As a second example, we consider the push-top axiom.

\text{stack}(\text{init}.h.\text{push}(a).\text{top}) = a.

To verify that our implementation satisfies this axiom, we must prove the following formula in Hoare’s
logic, where y is a variable of type itemtype and h is an arbitrary sequence of O-operations.

v: \{true\} \text{init(); } h; \text{push(a); } y=\text{top()} \{y=a\},

where h is an arbitrary sequence of operations. We replace each method by its body, replacing in top()
return statements by assignments to y and replacing in the body of push the formal parameter by a, which
yields:

\{true\}

\text{sindex:=0;}

h;

\text{sindex:=sindex+1; sarray[sindex]:=a;}

if (sindex>0) \{y := sarray[sindex];\} else \{\text{return error;\}

\{y=a\}

Because the code in this formula is a sequence, we need to apply the sequence statement rule; we need to
apply it three times. The question we must ponder, of course, is what intermediate assertion to put after
sequence h, given that we do not what h is. We submit: \{sindex\geq0\}. This yields the following formulas:

v0: \{true\} sindex:=0 \{sindex=0\}.

v1: \{sindex=0\} h \{sindex\geq0\}.

v2: \{sindex\geq0\} sindex := sindex+1 \{sindex>0\}

v3: \{sindex>0\} sarray[sindex]:=a \{sindex>0 \land sarray[sindex]=a\}

v4: \{sindex>0 \land sarray[sindex]=a\}

if (sindex>0) \{y := sarray[sindex];\} else \{\text{return error;\}

\{y=a\}
Application of the assignment statement rule to $v_0$, $v_2$, $v_3$ yields, respectively:

$v_{00}$: $\text{true} \Rightarrow 0=0$,

$v_{20}$: $sindex \geq 0 \Rightarrow sindex+1 > 0$,

$v_{30}$: $sindex > 0 \Rightarrow sindex > 0 \land a = a$,

all of which are tautologies. We consider $v_1$, to which we apply the consequence rule:

$v_{10}$: $\{sindex \geq 0\} \ h \ \{sindex \geq 0\}$.

In order to establish the validity of this formula, where $h$ is a sequence of operations, it is sufficient (by virtue of induction) to prove it for each individual operation; it is certainly valid for $V$-operations, since by definition they do not modify state variables (viz. $sindex$). As for $O$-operations,

- **Init**: The formula $\{sindex \geq 0\} \ init() \ \{sindex \geq 0\}$ is valid by virtue of the consequence rule, since $\{true\} \ init() \ \{sindex=0\}$ is valid.
- **Push**: The formula $\{sindex \geq 0\} \ push(a) \ \{sindex \geq 0\}$ is valid since the push operation increases the value of $sindex$.
- **Pop**: The formula $\{sindex \geq 0\} \ pop() \ \{sindex \geq 0\}$ is valid since the pop operation does not decrease the value of $sindex$ unless $sindex$ is positive, and then it is only decreased by 1.

Now we focus on formula $v_4$, to which we apply the if-then-else rule. We find,

$v_{40}$: $\{sindex > 0 \land sarray[sindex]=a \land sindex > 0\} \ y := sarray[sindex]; \ \{y=a\}$

$v_{41}$: $\{sindex > 0 \land sarray[sindex]=a \land sindex \leq 0\} \ y := sarray[sindex]; \ \{y=a\}$

The formula $v_{41}$ is vacuously valid since the precondition is false. We apply the assignment statement rule to $v_{40}$:

$v_{400}$: $sindex > 0 \land sarray[sindex]=a \Rightarrow sarray[sindex]=a$.

This formula is a tautology, hence an axiom of Hoare’s inference system; this concludes our proof. Continuing in this manner, we can establish that our code is correct with respect to all the axioms.

In practice, by the time they have gone through this phase, students have weeded out most of the trivial faults that may lie in their code, and they find it instructive that they were able to debug their program without ever executing it; as for subtle/complex faults, they may uncover them during testing.

### 5. Testing

So far we have used Hoare logic to prove that our implementation is correct with respect to the axioms of the specification; proving that it is also correct with respect to the rules is very difficult, hence we turn to testing. A testing policy is based on three decisions: how to generate test data; how to generate/develop an oracle; and how to analyze data. We review these questions in turn, below:
• **Test Data Generation.** We feel that usage-pattern based random test data generation offers the best tradeoff in terms of effort (test data is generated automatically), thoroughness (we can run millions of tests in a single experiment), and coverage (by generating test data according to the usage pattern, we detect the most egregious faults first). Note that because of the form of our specifications, test data takes the form of input histories, where each history is a sequence of method calls.

• **Oracle Generation.** We resolve to use rules as oracles; in other words, we check through testing whether our implementation satisfies the rules of the axiomatic specification. Many of these rules involve sequences of method calls (\(h, h', h+\), etc); these are generated randomly, as we discuss above. Also, many of the rules express an equivalence between module states, which raises the question: what does it mean for two states to be equivalent. We adopt the following answer: We consider that two states \(s\) and \(s'\) are equivalent (for the purposes of our test) if each \(V\)-operation returns the same value in state \(s\) and in state \(s'\).

• **How to Analyze Test Data.** For the purposes of this phase, we are only interested to record the failure rate of the module when it is submitted to a large set of test data. We are not removing any faults in this phase, only recording how often the module execution satisfies the oracle, and how often it fails to. In a way, this phase is not testing our ADT code as much as it is testing the test driver: it is really ensuring that our test driver is reliable, in preparation for the next phase (section 6), when we use the test driver to remove faults and estimate the resulting system reliability.

In light of these decisions, the outermost structure of our test driver look as follows:

```cpp
def failure_rate()
{
    int nbf=0;  // number of unsuccessful tests
    for (int i=0; i<testsize; i++)
    {
        switch (i%9)
        {
            case 0: initrule();   case 1: initpoprule();
            case 2: pushpoprule();  case 3: sizerule();
            case 4: emptyrulea();  case 5: emptyruleb();
            case 6: vopruletop();  case 7: voprulesize();
            case 8: vopruleempty();
        }
        cout << "failure rate: "  << nbf << "out of " << testsize << endl;
    }
}
```

This loop will cycle through the rules, testing them one by one successively. The factor `testsize` determines the overall size of the test data; because test data is generated automatically, this constant can be arbitrarily large, affording us an arbitrarily thorough test. The variable `nbf` represents the number of failing tests, and is incremented by the routines that are invoked in the switch statement, whenever a test fails. For the sake of illustration, we consider the `pushpoprule()`, which we detail below:
void pushpoprule()
{
  // stack(init.h.push(a).pop.h+) = stack(init.h.h+)
  int hl, hop[hlength]; itemtype hparam[hlength]; // storing h
  int hplusl, hplusop[hlength]; itemtype hplusparam[hlength]; // storing h+
  int storesize; boolean storeempty; itemtype storetop;

  // Variables used to store V-op values from the left side of the rule
  boolean successfultest; // outcome of the test

  // drawing h at random, storing it in hop, parameters in hparam
  hl = randnat(hlength); // drawing the length of h at random
  for (int k=0; k<hl; k++) // filling hop
    {hop[k]=gt0randnat(Xsize); // drawing k-th operation
      if (hop[k]==2) {hparam[k]=gt0randnat(paramrange);}} // param for push

  // same thing, for h+
  hplusl = gt0randnat(hlength);
  for (int k=0; k<hplusl; k++)
    {hplusop[k]=gt0randnat(Xsize);
      if (hplusop[k]==2) {hplusparam[k]=gt0randnat(paramrange);}}

  // left hand side of rule
  s.sinit(); historygenerator(hl,hop,hparam); // init.h
  itemtype a=randnat(paramrange); s.push(a); s.pop(); // push(a).pop
  historygenerator(hplusl,hplusop,hplusparam); // h+

  // store resulting state
  storesize = s.size(); storeempty=s.empty(); storetop=s.top();

  // right hand side of rule
  s.sinit(); historygenerator(hl,hop,hparam); // init.h
  historygenerator(hplusl,hplusop,hplusparam); // h+

  // compare current state with stored state
  successfultest =
    (storesize==s.size()) && (storeempty==s.empty()) && (storetop==s.top());

  if (! successfultest) {nbf++;} // counting failures
}

We have included comments in the code to explain it. Basically, this function proceeds as follows: First, it generates histories $h$ and $h+$; then it executes the sequence $\text{init.h.push}(a).\text{pop.h+}$, for some arbitrary item $a$; then it takes a snapshot of the current state by calling all the $V$-operations and storing the values
they return. Then it reinitializes the stack and calls the sequence `init.h.h+`; finally it verifies that the current state of the stack (as defined by the values returned by the V-operations) is identical to the state of the stack at the sequence given in the left hand side (which was previously stored). If the values are identical, then we declare a successful test; if not, we increment `nbf`.

Execution of the test driver on our stack yields the following outcome:

**failure rate: 0 out of 10000**

which means that all 10000 executions of the stack were consistent with the rules of the stack specification. Of course, typically, when we are dealing with large and complex module, a more likely outcome is to observe a number of failures. In the next section we discuss how we propose to organize the fault removal process, and how we use this process to estimate the module’s reliability.

### 6. Reliability Estimation

We adopt the discipline advocated by the Cleanroom [Mills et al, 1987; Prowell et al, 1999] software engineering methodology, which provides the following measures:

- Test data is generated at random, according to a predefined usage pattern (although we did not worry about the usage pattern aspect in the previous section ---our test data was uniformly distributed). In practice we can represent usage patterns by means of matrices that represent transition probabilities between method calls.
- Static verification (in the style we have advocated in section 4) replaces unit testing; all executions of the software product, from the very first, are done under public scrutiny, and are duly documented and recorded in the context of the reliability estimation.
- Testing is carried out against a formally defined oracle, and all reliability calculations and claims are made relative to the selected oracle.

Cleanroom’s reliability estimation is based on the following equation,

\[
MTTF_N = MTTF_0 \times R^N,
\]

where

- \(MTTF_0\) is the mean time to failure (measured as the estimated number of executions before the next failure) of the software product at the beginning of the testing phase,
- \(MTTF_N\) is the mean time to failure after \(N\) faults have been removed,
- \(R\) is a reliability growth factor, that is normally greater than 1.

The testing phase proceeds as follows:

- We use the test driver discussed in the previous section, but with a slight modification: instead of proceeding with the test for a fixed number of times (in a for loop), we iterate only as long as there are no failures (each rule that is tested returns true).
• When a rule is violated, we stop the test, analyze the data on which the product failed, identify the fault that caused the failure, and record the number of tests since the last failure. The scenario made up of testing the software product iteratively it fails, then removing the fault that is suspected of causing the failure is called a test run.

After we have proceeded through \( N \) test runs, we can estimate that the mean time to failure of the software product as the estimated length of the next run, i.e.

\[
MTTF_{N+1} = MTTF_0 \times R^{N+1}.
\]

At the end of \( N \) runs, we know what \( N \) is, of course; we need to determine the remaining constants, namely \( MTTF_0 \) and \( R \). To determine these two constants, we use the historic data we have collected on the \( N \) first runs, and we take a linear regression on the logarithmic version of the equation above:

\[
\log (MTTF_N) = \log (MTTF_0) + N \times \log(R).
\]

We perform a linear regression where \( \log (MTTF_N) \) is the dependent variable, and \( N \) is the independent variable. For the sake of argument, we show in the table below a sample record of a reliability test, where the first column shows the ordinal of the runs and the second column shows the length of each run (measured in terms of the number of executions before failure). Our stack ran without failure for 10000 tests, but that is because the implementation is very simple, the code is very short, and we have already proven the correctness of its code with respect to the axioms of the specification (and removed one fault we had in the push operation). But it is quite conceivable that with larger and more complex code, the verification step may leave a number of residual faults in the source code.

<table>
<thead>
<tr>
<th>( N )</th>
<th>Inter-Failure Run</th>
<th>Log</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>24</td>
<td>1.330211</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
<td>1.30103</td>
</tr>
<tr>
<td>2</td>
<td>36</td>
<td>1.556303</td>
</tr>
<tr>
<td>3</td>
<td>400</td>
<td>2.60206</td>
</tr>
<tr>
<td>4</td>
<td>510</td>
<td>2.70757</td>
</tr>
<tr>
<td>5</td>
<td>10000</td>
<td>4.0</td>
</tr>
</tbody>
</table>

In the third column, we record the logarithm of the inter-failure test runs. When we perform a regression using the third column as dependent variable and the first column as independent variable, we find the following result:

\[
\log \text{MTTF} = 0.946125044501616 + 0.524694909237272 \times N.
\]

Figure 1 (below) shows the quality of the regression, on a logarithmic scale. From this equation we infer the missing constants in our reliability equation, namely

- \( MTTF_0 = 10^{0.9461250445} = 8.833342 \).
- \( R = 10^{0.524694909} = 3.34730209 \).
From which we infer the mean time to failure at the conclusion of the testing phase as follows:

\[ MTTF = 8.833342 \times 3.34730209^6 = 12425. \]

In other words, when this software product is delivered, it is expected to execute 12425 times before its next failure.

We try to impress on students the concept of software warranty, whereby they deliver a software product with a formal claim about its reliability, supported by empirical observations of the behavior of the product under fully documented operating conditions. This is a far cry from statements that students are prone to make about their programs, such as: this program is correct, we just tested it and it works.

7. Conclusion

In this paper, we have presented an integrated software lifecycle that supports an agile/ lightweight methodology that combines formality / precision with ease of use and intuitive appeal. This lifecycle is focused on the specification, implementation and analysis of software modules whose behavior is dependent on input history, but can of course be applied to products which define a simple input/ output mapping. The proposed lifecycle proceeds through the following phases:
• **A Specification Phase,** where the product in question is specified by means of a relation linking its input history to its output; these specifications are written inductively, with axioms specifying the behavior of the module for simple histories and the rules linking the behavior on complex histories to their behavior on simpler histories.

• **A Specification Validation Phase,** where the specification that is generated by the specifier team is confronted against validation data produced by the V&V team to ensure the completeness and (to some extent) the minimality of the generated specification.

• **An Implementation Phase,** where a software product is produced to satisfy the axiomatic specification. Because the specification model used is behavioral rather than model-based, it leaves maximal latitude to the implementer in making design and implementation decisions.

• **A Verification Phase,** where the software product is verified against the axioms of the specification using Hoare logic. Because the axioms refer to elementary behavior of the product, they generally involve simpler sections of the source code, and obviate the aspects that generally make program proofs complex (such as the need to generate loop invariants, complex intermediate assertions, complex weakest preconditions, complex strongest postconditions, etc).

• **A Testing Phase,** where we develop a test driver that automatically generates random test data and repeatedly tests the implementation against oracles that are derived from the rules of the specification. We perform no fault removal (debugging) in this phase; paradoxically, the main purpose of this phase is really to ensure that the test driver is written correctly, in preparation for the next phase.

• **A Reliability Estimation Phase,** in which the previously developed test driver is slightly retooled and deployed to remove faults and record the evolution of the failure rate, until the resulting mean time to failure reaches the desired target.

At the conclusion of this six-phase lifecycle, the engineer has much to show for her/his effort:

• A validated formal specification for her/his product,

• An implementation that has been formally verified against some aspects of the specification (dealing with base cases) and thoroughly tested against others (dealing with inductive cases).

• An empirical estimation of the reliability of the product with respect to the formal specification at hand.

• The discipline of developing software products with associated claims about their quality, claims that the student/engineer is prepared to stand behind since they are based on empirical observations coupled with sound analysis.

This methodology has been used in a classroom environment, and students are generally able to relate to it and use it effectively within a fifteen-week term, without the usual gripes about formal methods. As far as future plans are concerned, we are interested to develop tools that support the main phases and activities of this methodology.

8. **Bibliography**


