Validating Software via Abstract State Specifications

Jonathan S. Ostroff, *Senior Member, IEEE,*

***Abstract*—We describe two tools—ETF and Mathmodels—for developing reliable software by eliciting precise specifications, validating them and verifying that the final software product satisfies the requirements. Mathmodels extends the classical Eiffel contracting notation with the use of mathematical models (sets, sequences, relations, functions, bags) to describe abstract state machines. Classical contracts are incomplete or are low level implementation assertions. Mathmodel contracts provide complete specifications of components and systems that can be verified via runtime contract checking scaling up to large systems. Mathmodels are void safe and have immutable queries (for specifications) as well as relatively efficient mutable commands for the abstract description of algorithms.**

The ETF tool is used in requirements elicitation to derive specifications, to describe the user interface, to identify the abstract state, and to develop use cases before the software product is constructed. The ETF tool generates code that decouples the user interface from the design (the business logic). The ETF Tool supports the derivation of important system safety invariants which become Mathmodel class invariants in the production code. The ideas can be extended to other contracting languages and frameworks and are placed in the context of best practices for software engineering. We also discuss this work in the light of proposals for software engineering education.

***Index Terms*—requirements, specifications, programs, software verification, tools, formal methods, reliable software, software engineering**

CONTENTS

Jonathan S. Ostroff, P.Eng, Ph.D, is Professor in the Department of Electrical Engineering and Computer Science in the Lassonde School of Engineering at York University, in Toronto, Ontario, Canada. Email: jonathan@yorku.ca.

I. INTRODUCTION

ENGINEERS and architects draw detailed plans before a brick is laid or a nail is hammered. Programmers and software engineers don't. Can this be why buildings and bridges seldom collapse and programs often crash? Blueprints help architects ensure that what they are planning to build will work. "Working" means more than not collapsing; it means being safe and serving the required purpose. Architects and their clients use blueprints to understand what they are going to build before they start building it. But few programmers write even a rough sketch of what their programs will do before they start coding. (Leslie Lamport [1]).

As Lamport writes, a blueprint for a program is called a specification. Architects and engineers find blueprints useful, even though buildings and machines cannot be automatically generated from them. So too, specifications are useful. A specification is an abstraction. It should describe the important aspects and omit the unimportant ones, an art that is learned only through practice. The specification should describe everything one needs to know to use the code. It should never be necessary to read the code to find out what it does.

Specifications allow us to think above the code level so as to understand our programming task at a higher level before we start writing code (by ignoring distracting implementation detail). It is a good idea to think about what we are going to do before doing it, and as the cartoonist Guindon wrote: "Writing is natures way of letting you know how sloppy your thinking is" [1].

Writing specifications will not catch all errors, and thus there will still be a need to test and debug to find them. A formal specification requires more up front thinking yet does not guarantee that one will not make mistakes. But not thinking guarantees that one will. With effort, specification can save time by catching requirement and design errors when they are easier to fix, before they are embedded in code.

Not all programs are worth specifying formally. The main reason for writing a formal specification is to use tools to validate it. Tools cannot find design errors in informal specifications.

The effort of formal specification may only be useful for complex components of the software product. However, even in formal specification there are a range of methods and tools that allow us to apply formal methods in a lightweight manner.

A variety of software specification methods are described in [2] from the very formal to lightweight methods. At the very formal end, experience with interactive theorem proving has shown that the cost of proofs of correctness is usually an order of magnitude greater than the cost of specification itself, requiring special expertise. Full formalization in very expressive languages may thus be necessary in safety critical systems. But in ordinary mission critical business systems, a lightweight approach that emphasizes partiality and focused application can bring great benefits at reduced cost.

A. *Models, Seamlessness and Reversibility*

Model-based specification is an approach to specifying software components where the system specification is expressed as a state model. This state model is constructed using well-understood mathematical entities such as tuples, sets, functions, relations and bags. System operations are specified by defining how they affect the state of the system model. There are many widely used notations for developing model-based specifications such as VDM, Z, B, Event-B and TLA+. The monograph “Software Specification Methods” [2] compares the various state bases approaches as well as alternative methods such as algebraic specifications.

A fundamental issues in all these approaches is *validating* the specifications to ensure that the specification matches the client’s needs. The specification must also be checked for completeness and consistency. To do this tools are needed beyond that of syntax and type checkers. A range of tools from the very formal to lightweight methods have been devised including theorem provers, modelcheckers, and SMT solvers. The more formal the methods, the greater the expertise needed to use the appropriate tools.

Beyond validation there is the issue of *verification*, i.e. checking that an implementation in a modern programming language or framework satisfies the specification. A common approach is to generate test cases from the specification to check the correctness of the implementation.

The challenge: In going from requirements analysis, to specifications, design and implemented code there is a significant challenge. There is usually no smooth transition from requirements to working code.

Model-Driven Engineering (MDE) is an approach to system engineering that uses models as an integral part of requirements, analysis, design, implementation, and verification of a system or product throughout the development life cycle. The main proposal is to focus on models rather than on computer programs generating lower-level models, and eventually code, from higher-level models [3]–[5]. These models are based on a variety of methods and often include graphical constructs with a formal syntax and semantics such as UML and SysML [6].

There are substantial challenges in model driven engineering. “The complexity of languages such as the UML is reflected in their metamodels. Complex metamodels are problematic for developers who need to understand and use them. These include developers of MDE tools and transformations. The complexity of metamodels for standard languages such as the UML also presents challenges to the groups charged with evolving the standards. An evolution process in which changes to a metamodel are made and evaluated manually is tedious and error prone. Manual techniques make it difficult to (1) establish that changes are made consistently across the metamodel, (2) determine the impact changes have on other model elements, and (3) determine that the modified metamodel is sound and complete. It is important that metamodels be shown to be sound and complete. Conformance mechanisms can then be developed and used by tool vendors to check that their interpretations of rules in the metamodel are accurate” [3].

The authors of graphical modelling tools usually do not compare their approaches to textual versions. The implicit assumption is that graphical representations are better simply because they are graphical. These notations must be provided with a formal semantics and transformation rules to executable code. If the code needs to be changed there are issues as to how this will be reflected back to the graphical model [5], [7].

If a specification notation is to be easily translated into a programming language, then the notation just becomes another procedural language (often introducing more complexity than it solves). Or, a completely different high-level specification language is invented that causes a semantic gap between specifications and code [8].

Seamlessness and reversibility are the critical issues in model based engineering. Seamlessness means that we can use the same notation for a large part of the development reducing the semantic gaps between the various tasks. Reversibility means that the seamless process must work in both directions. If one modifies an implementation, it must be possible to reflect the modification back to higher levels of design and specification.

B. *Classical Design by Contract (DbC) and Reliability*

Design by contract (DbC) allows software designers to specify precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and class invariants [9]. These specifications are referred to as “contracts”, in accordance with a conceptual metaphor with the conditions and obligations of business contracts. DbC avoids the impedance mismatch between specifications and implementations as contracts are written in the same syntax as programming language expressions. DbC thus contributes to seamlessness and reversibility [10].

Meyer defines reliable software as software that is correct, robust, and secure. Correctness is a system’s ability to perform according to its specification in cases of use within that specification. Robustness is a system's ability to prevent damage in cases of erroneous use outside of its specification. Security is a systems ability to prevent damage in cases of hostile use outside of its specification.

Software correctness is a relation between code and a specification of the expected behaviour of the software component or product. Without proper specifications, correct software cannot be defined. The contracting method is a way to tightly integrate specifications into software development for documentation, understanding object-oriented inheritance, runtime assertion checking, and automated testing.

Writing software specifications still seems to be "disliked by almost everyone" [11]. Part of this is the perceived high cost/benefit ratio of writing and maintaining accurate specifications on top of the code. Developers will be more inclined to write specifications as long as they are simple, have a straightforward connection with the implementation, and help to debug code better and faster. Classical DbC supports simple executable specifications, written in the same syntax as programming language expressions. It supports design, incremental development, and testing and debugging. Experiences with this technique shows that providing lightweight specifications is an accepted practice when it brings tangible benefits and integrates well with the overall development process.

C. Contributions of this paper

The waterfall method is a (non-iterative) design process used to develop software products, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of requirements, analysis, design, construction, testing, and maintenance.

One of the problems with the waterfall method is that in many systems eliciting requirements is often the most difficult part of software development [12]. Customers are not always able to visualize an application from a requirements document. Another potential drawback is the possibility that the customer will be dissatisfied with their delivered software product and change is inevitable.

In consequence, agile methods have been adopted which is an iterative, team-based approach to development [13], [14]. An important contribution of the agile approach is the rule of associating a test with every piece of functionality and regression testing as more functionality is added. At the same time there are limitations to testing alone; as Edsger W. Dijkstra has pointed out that testing shows the presence of bugs but never their their absence. Contracting and testing together provide more confidence in the reliability of the software product than either one on their own [15].

Below, we list the contribution of the Mathmodels Library and ETF (Eiffel Testing Framework) Tool to the production of reliable software in the context of various software engineering tasks such as requirements, specifications, design and code construction.

- In section II, we describe the use of the Mathmodels library for specifications. A specification of a system or a component of a system uses mathematical models (sets, sequences, relations, functions, bags) to describe an abstract state machine using contracts (preconditions, postconditions, and class invariants) in the Eiffel programming language. Classical contracts are incomplete or are low level implementation assertions. Mathmodel contracts provide complete specifications of components and systems that can be verified via runtime contract checking that scales up to verifying large systems. Mathmodels are void safe and have immutable queries (for specifications) as well as relatively efficient mutable commands for the abstract description of algorithms.
- In section III, we use the commands of Mathmodels to specify algorithms at a higher level of abstraction than regular programming. Lamport uses the Quicksort algorithm example to emphasize the difference between algorithms and programs, and to criticize the undue attention devoted to programming languages over abstract specifications. The same example in Mathmodels suggests that a good contracting language, equipped with the right abstraction mechanisms, can be effective at describing not only final implementations but also abstract algorithms. These abstract descriptions are also executable, at the possible price of non-optimal performance. The transformation to an optimal version can happen entirely within the same method and language, thus contributing to seamlessness and reversibility.
- In section IV, we describe how the ETF tool is used in requirements elicitation to derive a specification. We use a small case study, which we call EHealth, to motivate the development. EHealth is an electronic health system to ensure that patients medication prescriptions are safe. The ETF tool is used to elicit requirements, to specify the user interface, to identify the abstract state, and to develop use cases before the software product is constructed. The ETF tool generates code in Eiffel that decouples the user interface from the design (the business logic). The same use cases are used as acceptance tests when the final product is completed.
- In section V, we validate the consistency, completeness and safety of the specifications using the TLC model checking tool. An example of a system safety invariant is: no prescription for any patient shall have dangerous interactions between medications. Such system invariants are invaluable for ensuring the safety of mission critical systems. The specification language that TLC can check is a subset of the TLA+ language. Each construct in this untyped subset has an analogue in the Mathmodels typed language, although the Mathmodel constructs are more verbose. Thus, the semantic gap in going from TLC to a Mathmodels specification is relatively small as TLC and Mathmodels both specify via mathematical descriptions using sets, sequences, functions, relations and bags. TLC models are not used on the complete model, only on the critical parts of the specification to validate that the specification satisfies important safety invariants.
- In section VI, we complete the ETF generated code for the business logic with Mathmodel specifications derived from the requirements. The use cases (from the earlier

TABLE I
AN EIFFEL ABSTRACT STATE MACHINE FOR A STACK

```
class interface
  STACK [G]
create
  make

feature -- model
  model: SEQ [G] -- abstraction function

feature -- queries
  count: INTEGER -- number of items in stack
    ensure Result = model.count

  top: G -- top of stack
    require count > 0
    ensure Result ∼ model [1]

feature -- commands
  pop -- pop top of stack
    require count > 0
    ensure model ∼ old model.tail

  push (x: G) -- push ‘x‘ on to the stack
    ensure model ∼ (x ◁ old model)

invariant
      model.count = implementation.count
      -- implementation not shown
end
```

phase) may directly be used for acceptance testing of the software product. As the acceptance tests are run, the Mathmodel contracts are checked thus verifying the correctness of the design. There is a trace from numbered atomic requirements to the Mathmodel contracts. System safety invariants are thus encoded in the Mathmodel specifications. In this way, the program text retains important system consistency and safety properties, traced back to the original requirements.

Finally we compare our Mathmodels Library and ETF Tool with other approaches to the development of reliable mission critical business systems. The use of the ETF Tool and the Mathmodels Library for the production of reliable software scales up to very large systems as contract checking is done automatically at runtime.

II. MATHMODELS FOR SPECIFICATIONS

In this section, we introduce a specification library called *Mathmodels* to improve seamlessness and reversibility in the development of reliable software. The Mathmodels library has classes for tuples, sets (see Table VI), functions, sequences, relations (see Table XV) and bags. It is used for model-based specification—an approach to specifying software components where the system specification is expressed as a state model.

Mathmodels is developed in Eiffel which is a modern object-oriented language with built-in constructs for DbC. The basic ideas would work together in any language supporting DbC.[1]

A. Abstract State Machines

A well-known approach to formal specification is algebraic specification. A stack in generic parameter G is described as an abstract datatype having operations (mathematical functions or partial functions) and axioms. The axioms allow us to deduce the behaviour of the stack free of implementation detail. An object-oriented class may implement this datatype. Consider the Eiffel class *STACK* whose interface (contracts but no implementation) is shown in Table I.

Without the Mathmodels library, classical contracting only partially specifies the stack behaviour. Consider the feature *pop* in the stack class (ignoring, for the moment, the query *model*).[2] The precondition is classical because it can be written using the queries (in this case *count*). Classically, we cannot write a meaningful postcondition at the abstract level of a specification.

Once an implementation for the stack is provided (e.g. a linked list), then we can provide complete contracts in terms of the implementation. But these implementation contracts are "polluted" with code detail (such as the previous link of a node) that are irrelevant to the abstract specification.

To provide a complete abstract specification, we may use the Mathmodels library sequence class *SEQ*[*G*] to provide an abstract state model for a stack. We do this by declaring a query *model* of type *SEQ*[*G*]. Then the postcondition of *pop* asserts that the new *model* is equal to the tail of the old model sequence. Likewise the postcondition of *push* is the old *model* prepended with the new item *x*.[3]

Query *prepended* (with infix notation "◁") in Table I is a function routine that returns a new sequence the same as the old one prepended with the argument x. It has no side effects on the current model sequence of the stack. The queries of Mathmodel classes such as *SEQ* act like mathematical functions. Hence they may safely be used in contracts following the Command-Query separation principle.[4]

B. Abstraction Function

When specifying the stack, query *model* may be an attribute (not needing any implementation), and all contracts are specified in terms of the *model* sequence.

Class *STACK* may be implemented in a variety of ways, e.g. with an array or a linked list. Once an implementation for the stack is chosen, then query *model* must be given a body

[1]For documentation of Mathmodels, see http://www.eecs.yorku.ca/course_archive/2016-17/W/3311/eiffel-docs/mathmodels/index.html. The Library is available as open source at https://svn.eecs.yorku.ca/repos/sel-open/mathmodels.

[2]In Eiffel, a feature is either a *command* routine or a *query*. A command changes the state of the current object, while queries have no side effects on the current object but return a value. A query can either be an attribute (a variable) or a function routine.

[3]In the Table, for simplicity of presentation, we write the postcondition as *model* ∼ *x* ◁ **old** *model*). In Eiffel, the postcondition is written in ASCII: *model* ∼ **old** *model.deep_twin* |<− *x*
where ∼ is the symbol for object equality. We take the deep twin of the old model to ensure that there is no issue with aliasing (omitted for brevity in the Table). Eiffel syntax allows for an infix operator "|<−" (in ASCII) as an alias for the prepended query in class *SEQ*. In Eiffel, given two reference variables *v1* and *v2*, *v1* = *v2* is used for reference equality and *v1*∼*v2* is the Eiffel symbol for object equality. In Mathmodel, we use a value semantics, and thus always compare objects using object equality.

[4]Queries return a result and do not change the observable state of the system (are free of side effects). Commands change the state of a system but do not return a value.

TABLE II
ABSTRACTION FUNCTION: *model*

```
implementation: ARRAY [G]

model: SEQ [G] -- abstraction function
  do
    create Result.make_empty
    from i := implementation.lower
    until i > implementation.upper
    loop
      Result.prepend (implementation[i])
      i := i + 1
    end
  end
```

that maps the the concrete implementation state (in terms of an array or linked list) into into a sequence representing the abstract state as shown in Table II where the implementation of the stack is in terms of an array. Whatever the implementation, the body of query routine *model* must return a sequence equivalent to the implementation. The command *prepend* Mathmodel class *SEQ* is used to deduce the model sequence.

In subsection II-D below we describe the difference between commands such as *prepend* of the Mathmodel class *SEQ* used in the body of the *model* function routine in Table II and queries such as *prepended*.

C. Specifications vs. Implementations

A software specification normally describes the set of services a system or component is expected to provide (e.g. *push*, *pop* and *top* in the simple case of the stack). It must be be precise so that it can act as a contract between the client and the supplier (understandable by both). A specification is an abstraction. It should describe the important aspects and omit the unimportant ones. The specification should describe everything one needs to know to use the code. It should never be necessary to read the code to find out what it does. Specifications allow us to think above the code level so as to understand our programming task at a higher level before we start writing code. Thus a specification describes what the system will do but not how it will do it. The Mathmodel contracts in Table I are thus a specification of the stack.

By contrast, the stack can be implemented in many different ways. In Table II, the stack is implemented by an array. But it may also be implemented by a linked list or other data structures. Implementations thus describe how the stack will perform its operations. Implementations can change, but the Mathmodel specifications remain the same. This is the power of abstraction.

The stack example in Table I provides a simple illustration of what we mean by seamlessness. Formal specification languages must meet the same challenges as programming languages such as defining a coherent type system, supporting abstraction and modularity, and providing a clear syntax and semantics. In the stack, we use the same notation (Eiffel in this case) to express specifications and implementations within the same syntactic and semantic universe. In an ideal world where requirements are fixed at the start, one might switch notations between specification and implementation. But in practice requirements, designs and implementations change, and a seamless process relying on a single wide spectrum notation makes it possible to go back and forth between levels of abstraction without having to perform repeated translations between levels.

Thus, should we change the implementation of the STACK to a linked list, then all the Mathmodel contracts remain the same. Only the abstraction function *model* must be changed to reflect the new mapping from concrete to abstract state. If all unit tests are written at the interface level in terms of the public features, then the tests also remain unchanged despite the change in implementation. Contract violations at runtime will signal inconsistencies between specifications and implementations.

D. Immutable Functions vs. Commands

Once an implementation for the stack is chosen, then function routine *model* must be given a body that maps the implementation into into a sequence representing the abstract state. If the implementation changes, then the body of *model* must be changed to reflect the new abstraction function, but all other contracts remain unchanged. The mathematical class *SEQ* itself contains:

- Side-effect free function routines such as *prepended* (infix notation "$\lhd$") used in contracts as shown in Table 1;
- Command routines such as *prepend* that change the state of the current sequence as shown in the body of *model* in Table II.

The commands of *SEQ* can also be used to implement the stack, although perhaps not as efficiently as standard arrays, lists, dictionaries etc..

The feature *prepend* (of class *SEQ*) is a command that changes the state of its target. The call `Result.prepend(argument)`, in the body of query *model* in Table II, is a command that changes the state of the result. Queries of Mathmodel classes have command analogues. Thus a Mathmodel class such as *SEQ* can be used for specifications (via its queries) and (high-level but relatively) inefficient implementations (via its commands), that can later be refined to more efficient code without changing the specifications.

We specify the behaviour of the stack as an abstract state machine (using a mathematical sequence for the state) rather than as an abstract datatype with axioms to define behaviour. The idea of specifying a system by writing down all its axioms (as in abstract datatypes) seems like a good approach. But in practice, it often hard to decide whether the specification is complete or what additional properties are or are not implied by the axioms. Thus, specifying systems using abstract state machines (as in Table I) have become more popular. In the case of languages that support DbC, we obtain the benefit of specifications and final implementation as efficient executable code in a modern object oriented language that can be tested to satisfy the specification.

III. Mathmodels and Abstract Algorithms

Bertrand Meyer (the creator of the Eiffel method) has discussed a version of the Quicksort algorithm at a higher level

TABLE III
PARTITION ALGORITHM

```
a: ARRAY [G → COMPARABLE]
  −− array to be sorted in generic parameter G
pivot: INTEGER −− set by partition routine
picked: INTEGER_INTERVAL −− used by sorting algorithm

partition (i, j: INTEGER)
  require −− i..j is a sub−interval of the arrays legal indices
    $i \geq a.lower$
    $i < j$
    $j \leq a.upper$
  do
    −− usual implementation of partition
  ensure −− expected effect of routine partition
    $pivot \geq i$
    $pivot < j$
    −− a[i..j] has been reshuffled so that elements in i..pivot are less than
    −− or equal to those in pivot+1 .. j
  end
```

TABLE IV
RECURSIVE QUICKSORT

```
sort (i, j: INTEGER)
  require
    $i \leq j$
    $i \geq a.lower$
    $j \leq a.upper$
  do
    if $j > i$ then −− precondition of partition holds.
      partition (i, j) −− split into two slices $s$ and $t$ such that $s \leq t$
      sort (i, pivot) −− recursively sort first slice.
      sort (pivot+1, j) −− recursively sort second slice.
    end
  end
```

of abstraction than either conventional recursive or iterative programs.[5] His purpose is to compare the Eiffel method with the TLA+ method developed by Leslie Lamport [16]. A sorting routine is relatively easy to specify. Finding an efficient implementation is the difficult part (if one has has not already studied sorting algorithms).

Lamport presents this example in a lecture in which he asks the audience to give a non-recursive version of Quicksort. Participants attempt to remove the recursion by making the stack explicit or looking for invertible functions in calls. But Lamport's point is that recursion is not at all fundamental in Quicksort. The recursive version is a specific implementation of a more general idea of divide-and-conquer.

A. An Abstract version of Quicksort in Eiffel

We review Meyer's presentation and then show that the classical Eiffel libraries are insufficient and Mathmodels is thus needed. Meyer expresses Lamport's version (which he calls it *Lampsort*) in Eiffel in the context shown in Table III.

Routine *partition* is implemented in the usual way. The algorithm consists of doing nothing if the array has no more than one element, otherwise performing a partition and then recursively calling itself on the two resulting intervals. The implementation can take advantage of parallelism by forking the recursive calls out to different processors (Table IV). But that only describes a possible implementation.

The true Quicksort is more general. The algorithm works on a set *non_sorted* of integer intervals i..j such that the corresponding array slices a[i..j] are the only ones possibly not yet sorted. The goal of the algorithm is to make the non-sorted set empty, since then we know the entire array is sorted. In Eiffel we might declare this non sorted set as

not_sorted: *SET* [*INTEGER_INTERVAL*]

as shown in Table V.

In Eiffel, the function yielding an integer interval is declared in the library class INTEGER using the operator "|..|", rather than just "..".

[5]https://bertrandmeyer.com/2014/12/07/lampsort/, accessed 19 June 2017. The video of Lamport's lecture is at https://channel9.msdn.com/Events/Build/2014/3-642, segment starting at 0:32:34.

TABLE V
LAMPSORT IN EIFFEL

```
 1  not_sorted: SET [INTEGER_INTERVAL]
 2
 3  from
 4    create not_sorted.make_one (a.lower |..| a.upper)
 5  invariant Inv
 6  until not_sorted.is_empty
 7  loop
 8    picked := not_sorted.item
 9    if picked.count > 1 then
10      partition (picked.lower, picked.upper)
11      not_sorted.extend (picked.lower |..| pivot)
12      not_sorted.extend (pivot + 1 |..| picked.upper)
13    end
14    not_sorted.remove (picked)
15  end
```

The Lampsort algorithm initializes *not_sorted* to contain a single element—the entire interval. At each iteration, it removes an interval from the set, partitions it if that makes sense (i.e. the interval has more than one element), and inserts the resulting two intervals into the set. It ends when *not_sorted* is empty as shown in Table V:

- Line 4: initialize interval set to contain a single interval, the arrays entire index range;
- Line 5: this is the loop invariant *Inv* (see below);
- Line 6: stop when there are no more intervals in the set;
- Line 8: pick an interval from the non-empty interval set. The query *item* from *SET*, with the precondition **not** *is_empty*, returns an element of the set. It does not matter which element. In accordance with the Command-Query Separation principle, calling *item* does not modify the set. To remove the element, use the command *remove*. The command *extend* adds an element to the set.
- Line 9: ensure the precondition of *partition*;
- Lines 11 and 12: insert new sub-intervals derived from the partion algorithm, into the set of intervals;
- Line 14: remove interval that was just partitioned.

According to Meyer, the abstract idea behind Lampsort, explaining why it works at all, is in the loop invariant *Inv*. See [17] for a more general discussion of how invariants provide the basis for understanding loop algorithms.

Let a slice of an array be a non-empty contiguous sub-array. We may concatenate adjacent slices. Also, for slices s and t, $s \leq t$ means that every element of s is less than or equal to every element of t. The invariant is as follows. The array a is the concatenation of a set *slices* of disjoint slices, such that

- The elements of a are a permutation of its original elements;
- The index range of any member of slices having more than one element is in *not_sorted*;
- For any adjacent slices s and t (with s before t), $s \leq t$.

The invariant also ensures that the call to the partition routine satisfies that routines precondition. Meyer leaves the invariant informal and does not show how it is specified in Eiffel, nor does he provide a loop variant needed to check termination.

The Lampsort algorithm is a simple loop. It does not use recursion, but relies on an interesting data structure, a set of intervals. It is not significantly longer or more difficult to understand than the traditional recursive version

Lampsort, in its author's view, captures the true idea of Quicksort. The recursive version, and its parallelized variants, are only examples of possible implementations.

B. Methodological Debate

Meyer summarizes as follows: "I wrote at the start that the focus of this article is Lampsort as an algorithm, not issues of methodology. Let me, however, give an idea of the underlying methodological debate. Lamport uses this example to emphasize the difference between algorithms and programs, and to criticize the undue attention being devoted to programming languages. He presents Lampsort in a notation which he considers to be at a higher level than programming languages, and it is for him an algorithm rather than a program. Programs will be specific implementations guided in particular by efficiency considerations. One can derive them from higher-level versions (algorithms) through refinement. A refinement process may in particular remove or restrict non-determinism, present in the above version of Lampsort through the query *item* (whose only official property is that it returns an element of the set)."

"The example of Lampsort in Eiffel suggests that a good language, equipped with the right abstraction mechanisms, can be effective at describing not only final implementations but also abstract algorithms. It does not hurt, of course, that these abstract descriptions can also be executable, at the possible price of non-optimal performance. The transformation to an optimal version can happen entirely within the same method and language."

It should be noted, however, that Lamport's TLA+ language and accompanying toolbox perform a different function than Meyer's Eiffel specification. The toolbox can be used to exhaustively modelcheck [18] the specification over a bounded domain using the TLC model checker. Also, for an expert willing to use the theorem prover, the TLA+ algorithm can be proved to satisfy its specification. However, theorem proving requires greater and effort and expertise.

By contrast, Eiffel can be used to efficiently implement the specifications and via runtime checking verify the specification. This scales up to very large systems. However, it does not have the exhaustive testing facilities of TLC. However, the specifications that TLC can check are more limited than the facilities provided by programming languages such as Eiffel.

TABLE VI
MATHMODELS: SET[G]

```
class SET [G] create
  make_empty
feature -- commands with efficient implementation
  choose_item -- choose an arbitrary element
    require not is_empty
    ensure has (item) and chosen
  remove_item
    require not is_empty and chosen
  extend (g: G)
    -- Extend the current set by 'g'.
  union (other: SET[G])
    -- Union of the current set and 'other'.
  subtract (g: G)
   -- Subtract the current set by 'g'.
  difference (other: SET[G])
    -- Difference of the current set and 'other'.
feature -- immutable queries
  chosen: BOOLEAN
  item: G -- an arbitrary member
    require not is_empty and chosen
  count alias "#": INTEGER
    -- Return the cardinality of the set.
  is_empty: BOOLEAN
    -- Is the set empty?
  has (g: G): BOOLEAN
    -- Does the set contain 'g'?
  extended alias "+" (g: G): SET[G]
    -- Return a new set representing the addition of 'g' to Current
  unioned alias "|\/|" (other: SET[G]): SET[G]
    -- Return a new set representing the union of Current and 'other'
  subtracted alias "−" (g: G): SET[G]
    -- Return a new set representing the subtraction of 'g' from Current
  differenced alias "|\" (other: SET[G]): SET[G]
    -- Return a new set representing the difference between Current and
       'other'.
  ...
end
```

The TLC specification is in terms of an array of integers (the model checker does not deal with real number or more complex structures). The Eiffel specification for Lampsort is expressed in terms of an array in generic parameter G that inherits from COMPARABLE. Thus it works and can be tested on reals and more general comparable structures.

C. The need for Mathmodels

Consider Meyer's version of the Lampsort algorithm in Table V. The base library of Eiffel for SET[G] does allow one to extend and remove elements. But, there is no support for queries (function routines) that might be used in preconditions, postconditions, and loop variants and invariants needed for correctness. Also, there is no support for quantification. In particular, the base library does not support a query *item* needed to pick an arbitrary item in the set.

The Mathmodels set library shown in Table VI does have the required support. If the set is not empty, the command *choose_item* ensures that *item* can be queried for an arbitrary element of the set.

The specification of Lampsort in Mathmodels is shown in Table VII. The algorithm is specified with a precondition and postcondition. The abstract algorithm is described using a set of disjoint array intervals. The algorithm is also provided with

TABLE VII
SPECIFICATION OF LAMPSORT USING MATHMODELS SET[G]

```
 1  a: ARRAY[G → COMPARABLE]
 2  p: INTEGER −− pivot index returned by partition
 3
 4  lampsort (i, j: INTEGER)
 5      −− sort slice of array ‘a[i..j]’ using pivot ‘p’
 6    require
 7      a.lower <= i and i <= j and j <= a.upper
 8      i <= p and p <= j
 9    local
10      not_sorted: SET [INTEGER_INTERVAL]
11      picked: INTEGER_INTERVAL
12    do
13      from
14        −−initialize interval set to contain a single interval i..j
15        create not_sorted.make_one (i |..| j )
16      invariant −− see below
17        Inv
18      until −− stop when there are no more intervals in set
19        not_sorted.is_empty
20      loop
21        −− pick an interval from nonempty set
22        not_sorted.choose_item
23        picked := not_sorted.item
24        −− remove interval that was just picked for partitioning
25        not_sorted.remove_item
26        if picked.count > 1 then −− precondition of partition holds
27          partition (picked.lower, picked.upper)
28          if picked.lower < p then
29            not_sorted.extend (picked.lower |..| (p − 1)) end
30          if picked.upper > p then
31            not_sorted.extend ((p + 1) |..| picked.upper) end
32        end
33      variant −− total number of unsorted indices
34       sum_of_interval_counts (not_sorted)
35      end
36    ensure
37      −− ∀k ∈ i · · · j − 1 : a[k] ≤ a[k + 1]
38      across i |..| j−1 as k all a [k.item] <= a [k.item + 1] end
39      permutation (a, old a)
40    end
41
42   ___________________________________________
43   −− loop invariant for the above routine
44  invariant Inv
45    permutation (a, old a)
46    i <= p and p <= j
47    −− partitions in intervals are disjoint
48    across not_sorted as it1 all
49    across not_sorted as it2 all
50      it1.item /∼ it2.item implies disjoint (it1.item, it2.item)
51    end end
52    −− a[1..n] is sorted iff all partitions in set intervals are sorted
53    sorted (i, j) =
54      (across not_sorted as it all
55        sorted (it.item.lower, it.item.upper)
56      end)
57
58  ___________________________________________
59   −− loop variant for the above routine
60   −− returns the total count of numbers in the intervals in the set
61  sum_of_interval_counts (intervals: SET [INTEGER_INTERVAL]):
62    INTEGER
63      do
64        across
65          intervals as interval
66        loop
67          Result := Result + interval.item.count
68        end
69      end
```

TABLE VIII
COMPARISON OF LAMPSORT VS. RECURSIVE SORT

No. of elements	Lampsort	Recursive
100k	0.12s	0.07s
1m	1.30s	0.90s
10m	13.8s	10.1s

Contract checking is turned off. 3Ghz Mac-Mini, 16Gb RAM
100k is an array with 100,000 integers; 1m is an array with 1 million integers; 10m is an array with 10 million integers

a loop invariant *Inv* and variant to verify termination and correctness. These contracts are checked at runtime every time the sort routine is executed.

The postcondition (line 37) asserts that the array slice $i..j$ is sorted. In mathematical notation that is:

$\forall k \in i..j{-}1 : a[k] \leq a[k+1]$

Universal and existential quantification in the Mathmodels collections uses Eiffel’s **across** construct (via a built-in iterator pattern). Line 39 asserts that the sorted array is a permutation of the initial array.

The loop invariant (from [19]) is shown in lines 43 to 56 of Table VII.[6] Lines 48 to 51 assert the following in mathematical notation

$\forall iv1, iv2 \in non\_sorted : iv1 \neq iv2 : disjoint(iv1, iv2)$

i.e. all the intervals in *non_sorted* are disjoint. The variant is just the total of the number of elements in each interval in *not_sorted*, as described in lines 61 to 69.
This variant is decreased by one or more each time through the loop.

Table VIII compares the efficiency of the Lampsort algorithm (using the Mathmodels set collection) with the standard recursive version of Quicksort (both use the same partition algorithm). As can be seen from the Table, the iterative Lampsort version (using Mathmodels *SET*[*G*]) is almost as efficient as the recursive version using just arrays. This comparison is done with contract checking turned off, in order to test the efficiency of the commands of *SET*[*G*]. Thus the abstract algorithm, when executed, is relatively efficient.

IV. ETF TOOL AND REQUIREMENTS ELICITATION

Validation of software products attempts to answer the question: are we building the right system? *Verification* answers the question: are we building the system right? Validation is the process of checking whether the requirements captures the customers needs, while verification is the process of checking that the software product satisfies the requirements.

In validation, we also attempt to show that the specification is consistent, and specifies a software product that is safe and fit for use. An important part of validation will thus be the derivation of important system safety invariants.

In this section we describe a tool, developed in Eiffel, called ETF (Eiffel Testing Framework). During requirements elicitation, the ETF Tool is used to derive a specification for the

[6]Line 44 is not supported in the current implementation of the Eiffel language, in which an invariant is a one state predicate. As a work-around, we use a local variable initialized to the original array in the loop initialization.

system under development and to validate the specfication.[7] The following are the goals of ETF:

- During requirements elicitation for business derive a testable specification for the system under development;
- Specify an abstract grammar for the user interface;
- Develop use cases (understandable by customers) before the product is developed;
- The use cases are also acceptance tests to validate the final software product;
- An abstract state (in the sense of Mathmodels) is derived from the use case analysis;
- Generate Eiffel code in which the the user interface of a system is decoupled from its business logic.

The kind of use case that we envisage is one that is closer to the notion of computations produced by abstract state machines, rather than the kind of Use Case considered in UML analysis [12]. The common element between our notion and UML, is that a use case must express a function or feature that has value to a customer external to the software product.

How should we describe such a use case which is an abstract computation of the computing device (the "machine") under development? Most computer scientists would probably interpret this question to mean, what programming language should we use? But programming languages are filled with unnecessary implementation detail that hide the abstract specification.

To describe an abstract computation we need to describe a sequence of events (the external user inputs) and states in consequence of the user inputs. We may describe a set of computations that can be produced by a some computing device by describing (1) the set of all initial initial states and (2) a next-state relation that describes, for every state, the possible next state, i.e. the set of states reachable from that state by a single step. Once engineers understand this notion of a computation is and how it is described, they can understand the importance of a system invariant. A computing device does the correct thing only because it maintains a correct state. Correctness of the state is expressed by an invariant—a predicate that is true in every state of every computation [20]. We will explore this aspect more in the next section. In this section, we describe how how ETF is used to to elicit the user inputs and an abstract state.

The ETF tool is independent of Mathmodels. However, Mathmodel descriptions are often used to specify the design of the business logic, and thus used in conjunction with ETF. Although ETF is written in Eiffel, the basic idea applies to any programming language and thus a suitable tool may be developed for any language or framework.

A. Requirements, Specifications and Programs

Jackson [21] writes that the terminology of software development is mostly in a chaos that correctly reflects the chaotic state of the field. Usage of the word "specification" is no exception. For precision, he uses the term in a more restricted way.

[7]ETF is open source and there is some documentation at http://seldoc.eecs.yorku.ca/doku.php/eiffel/etf/start.

Specification is one of a trio of terms: requirements; specifications; and programs. Specifications are all about—and only about—the shared phenomena at the interface between the machine (the computer) and the environment in which the machine must function. Requirements are all about—and only about—the environment phenomena. Programs, on the other hand, are all about—and only about—the machine phenomena.

The *machine*, in this context, is a computing device and its program that periodically takes inputs via user interfaces and sensors connected to the environment, and delivers outputs via actuators and displays.

We thus refer to the machine domain and the application domain (the environment in which the machine must operate). The machine is the thing that is to be built. The machine domain is the set of phenomena that the machine has access to: data structures it can manipulate, algorithms it can run, devices it can control, inputs it can get from the world, and outputs to the world (the environment). By contrast, the application domain is the world into which the machine will be introduced: it is that part of the environment in which the machines actions will be observed and evaluated.

Given that requirements engineering is concerned with the purpose of a system, requirements are part of the application domain, rather than the machine domain. It is the application domain that provides a purpose for the machine, and so it is the application domain that determines the requirements. The application domain and the machine domain must be connected somehow, because the machine must interact with the world in order to be useful. The connection is via shared phenomena—things that are observable both to the machine and to the application domain. Shared phenomena include events in the real world that the machine can directly sense (e.g. buttons being pushed, movements that sensors can detect) and actions in the real world that the machine can directly cause (e.g. images appearing on a screen, devices being turned on or off).

The specification is about those shared phenomena. A specification cannot be completely abstract because its ultimate subject matter is at the interface between the abstract and the concrete. If one relies on too much abstraction in the wrong places one's specification will be about an abstract problem, not about the real problem that the customer expects one to solve.

Let ENV stand for constraints and assumptions that come from the environment. Let REQ be the requirements, SPEC the specification and M for the machine implementation. Then validation and verification may be described by the following rule:

$$\boxed{\begin{array}{ll} ENV, SPEC \models REQ & \text{Validation of Spec} \\ M \models SPEC & \text{Verification of Design} \\ \hline ENV, M \models REQ & \text{System Correctness} \end{array}}$$

The environmental assumptions and constraints and the requirements describe the problem domain. The specification describes the solution domain. In validation, we check that the specification is describing a satisfactory solution, one acceptable to the customers and feasible from the point of view of implementation. In verification, we check that the implementation (the machine) satisfies the specification.

Recommend best practices for requirements include identifying the system boundary (between the machine and its environment), identifying the environmental assumptions and constraints, developing use cases, describing the software specifications, and allocating specifications to subsystems [22]. The authors of [22] write: “Use cases are a popular way to identify and document interactions between a system, its operators, and other systems. The literature on use cases is large and covers the gamut from high-level requirements down to detailed design. ... While conventions for the format of use cases are similar, there are numerous styles that introduce varying degrees of rigor and formality. However, use cases seem to be especially appropriate for use early in the REM (Requirements Engineering Management) process by helping to understand and describe how operators and other systems will interact with the system. One of their attractions is that they can be used relatively informally to elicit a better understanding of the System Functions the operators need. In effect, they provide early validation of the system behavior.”

B. Case Study: EHealth System

Writing a good requirement document is a difficult task. The readers of such a document are (a) customers who may not have technical knowledge and (b) the engineers and software developers who will conduct its specification and design. It is usually difficult for the engineers to exploit the requirement document if they cannot clearly identify what they have to take into account and in which order. Important points may be missing. For example, a large requirements document for the alarm system of an aircraft was missing the simple fact that this system should not deliver false alarms. When the authors of this document were interrogated on this missing point, the answer they gave was rather surprising: it was not necessary to put such a detail in the requirement document because “of course everybody knows that the system should not deliver any false alarm” [23].

On the other hand, the requirement document is sometimes over-specified with a number of irrelevant details. It is then difficult for the reader of the requirements document to distinguish between which part of the text is devoted to explanations and which part is devoted to genuine requirements. Explanations are needed initially for the reader to understand the future system. But when the reader is more acquainted with the purpose of the system, explanations are less important. At that time, what counts is to remember what the real requirements are in order to know exactly what has to be taken into account in the system to be constructed.

Our case study is an *EHealth* system which is an electronic health system where the goal is to ensure that there are no undesirable interactions between medications in patient prescriptions.

We follow Jackson and divide the descriptions into E-descriptions (environmental) constraints or assumptions and R-descriptions (what the machine must produce). Elicitation of informal requirements produces the following:

ENV1	Physicians prescribes medications to *patients*

ENV2	There exists pairs of medications that when taken together have dangerous *interactions*

For example, warfarin and aspirin both increase anti-coagulation.

ENV3	If one *medication* interacts with another, then the reverse also applies (Symmetry)

ENV4	A medication does not interact with itself (Irreflexive)

REQ5	The system shall maintain records of dangerous medication interactions

REQ6	The system shall maintain records of patient *prescriptions*. No prescription may have a dangerous interaction

REQ7	Physicians shall be allowed to add a medication to a patient’s prescription, provided it does not result in a dangerous interaction.

REQ8	It shall be possible to add a new medication interaction to the records, provided that it does not result in a dangerous interaction.

Thus, first remove the new dangerous interaction from patient prescriptions before adding the new interaction to the records.

REQ9	Physicians shall always be be allowed to remove a medication from a patient’s prescription.

The above requirements are informal and may be understood by customers and engineers alike. The requirements document is organized around two texts embedded in each other: the explanatory text and the reference text. These two texts should be immediately separable, so that it is possible to summarize the reference text (in the frames) independently. The reference

TABLE IX
EXAMPLE OF AN ETF GRAMMAR SPECIFICATION

system *ehealth*
−− manage prescriptions for physicians and patients

type *ID_MD* = *INT* −− physicians
type *ID_PT* = *INT* −− patients
type *ID_RX* = *INT* −− prescriptions
type *ID_MN* = *INT* −− medications

type *NAME* = *STRING*
  −− names of physicians, patients and medications

type *KIND* = {*pill*, *liquid*}
  −− for a pill, it is a positive real in mg.
  −− for a liquid it is a positive real in cc.

type *MEDICATION* =
  TUPLE [*name*: *NAME*; *kind*: *KIND*; *low*: *VALUE*; *hi*: *VALUE*]

type *PHYSICIAN* = {*generalist*, *specialist*}

−− User Actions
add_physician (*id*: *ID_MD*; *name*: *NAME*; *kind*: *PHYSICIAN*)
add_patient (*id*: *ID_PT*; *name*: *NAME*)
add_medication (*id*: *ID_MN*; *medicine*: *MEDICATION*)
add_interaction (*id1*:*ID_MN*;*id2*:*ID_MN*)

new_prescription (*id*: *ID_RX*; *doctor*: *ID_MD*; *patient*: *ID_PT*)
add_medicine (*id*: *ID_RX*; *medicine*:*ID_MN*; *dose*: *VALUE*)
remove_medicine (*id*: *ID_RX*; *medicine*:*ID_MN*)
...

text takes the form of labeled and numbered short statements written using natural language, which must be very easy to read independently from the explanatory text. The explanations are just there to give some comments which could help a first reader. But after an initial period, the reference text is the only one that counts [23].

Obviously, a real requirements document will contain many more numbered atomic descriptions organized hierarchically [24].

C. Using ETF to specify the user interface for EHealth

Table IX is an example of a specification of a grammar for the EHealth system. Based on the requirements, we specify a grammar for the user input to the system. One may use a variety of basic types such as INT, VALUE (arbitrary precision decimals), CHAR as well as tuples and sequences of tuples. New types can be formed from the basic types. For example, in Table IX, a type MEDICATION is defined as:

TUPLE [*name*: *NAME*; *kind*: *KIND*; *low*: *VALUE*; *hi*: *VALUE*]
We may also have arrays of tuples (recursively). Enumeration types such as KIND and PHYSICIAN are also supported.

Also defined, are possible user inputs such as adding medications, physicians, interactions, etc.

To keep the example manageable, we will use the grammar specification for the smaller system in Table X.

Table XI is an example of a Use Case for the EHealth system. In this use case, we add medications, physicians and dangerous interactions. We also prescribed medications for the various patients. The use case is a sequence of user actions

TABLE X
SPECIFICATION OF ETF GRAMMAR FOR EHEALTH

system *ehealth*
−− manage prescriptions for physicians and patients

type *MEDICATION* = *STRING*
type *PATIENT* = *STRING*

add_patient (*p*: *PATIENT*)
add_medication (*m*: *MEDICATION*)
add_interaction (*m1*: *MEDICATION*; *m2*: *MEDICATION*)
add_prescription (*p*: *PATIENT*; *m*: *MEDICATION*)
remove_interaction (*m1*: *MEDICATION*; *m2*: *MEDICATION*)
remove_prescription (*p*: *PATIENT*; *m*:*MEDICATION*))

and the abstract state after each user command at the user interface. If a user action is illegal, the system shall not crash or generate an exception. Rather a useful error message is provided to the user of the system. For example at state 17 we see an error report as shown in Table XII.

According to the abstract state, medication *m2* interacts with *m4*. Thus, a doctor cannot prescribe medication *m4* for patient *p3*, because this would be dangerous for the patient. This is because medication *m2* is already prescribed for the the patient.

This information is conveyed in the abstract state, in an ASCII format chosen by the requirements engineer so that non-technical customers can understand the use case as well. Thus `prescriptions: {p1->m1,m3; p3->m2}` in state 17 means that patient *p1* has been prescribed medications *m1* and *m2* and patient *p3* has ben prescribed medication *m2*.

The use case then continues as follows: The interaction $m2 \mapsto m4$ is removed at state 18. Then the prescription can be filled.

In a real example, we might need different kinds of physicians, e.g. generalists and specialists. Perhaps only specialists can prescribe a dangerous interactions. We have kept the example small for manageability in the context of this paper. As far as ETF is concerned, the grammar and use cases can be as complex as is needed. This is how the ETF Tool is used:

- Based on the requirements, specify a grammar for the user input to the system, e.g. as shown in Table X.
- The grammar specifies an abstract user interface. There is no need to commit, prematurely, to a concrete user interface.
- Using the grammar, write some use cases such as that shown in Table XI to be checked with the customer for validity. These use cases may be written before the software product is developed.
- The ETF tool is invoked on the grammar and generates code (as shown in Table XIII) that allows one to run the use cases from the command line.
- The input at the command line to the generated code is a sequence of events (satisfying the grammar) representing the customer interacting with the system. This will automatically generate an output (like that shown in Table XI but) with empty states at each step. If a user action is entered that does not satisfy the grammar, a syntax error

TABLE XI
ETF USE CASE: ADD DANGEROUS INTERACTIONS AND PRESCRIPTIONS

```
  state 0
  patients:      {}
  medications:   {}
  interactions:  {}
  prescriptions: {}
 ->add_patient("p1")
  state 1
  patients:      {p1}
  medications:   {}
  interactions:  {}
  prescriptions: {}
 ...
 ->add_patient("p3")
  state 3
  patients:      {p1,p2,p3}
  medications:   {}
  interactions:  {}
  prescriptions: {}
 ->add_patient("p3")
  state 4 Error e1: patient already entered
  patients:      {p1,p2,p3}
  medications:   {}
  interactions:  {}
  prescriptions: {}
 ->add_medication("m1")
  state 5
  patients:      {p1,p2,p3}
  medications:   {m1}
  interactions:  {}
  prescriptions: {}
 ...
 ->add_interaction("m1","m2")
  state 10
  patients:      {p1,p2,p3}
  medications:   {m1,m2,m3,m4}
  interactions:  {m1->m2,m2->m1}
  prescriptions: {}
 ->add_interaction("m2","m4")
  state 11
  patients:      {p1,p2,p3}
  medications:   {m1,m2,m3,m4}
  interactions:  {m1->m2,m2->m1,m2->m4,m4->m2}
  prescriptions: {}
 ->add_interaction("m2","m1")
  state 12 Error e3: interaction already added
  patients:      {p1,p2,p3}
  medications:   {m1,m2,m3,m4}
  interactions:  {m1->m2,m2->m1,m2->m4,m4->m2}
  prescriptions: {}
 ->add_prescription("p1","m1")
  state 13
  patients:      {p1,p2,p3}
  medications:   {m1,m2,m3,m4}
  interactions:  {m1->m2,m2->m1,m2->m4,m4->m2}
  prescriptions: {p1->m1}
 ...
 ->add_prescription("p3","m4")
  state 17 Error e4: this prescription dangerous
  patients:      {p1,p2,p3}
  medications:   {m1,m2,m3,m4}
  interactions:  {m1->m2,m2->m1,m2->m4,m4->m2}
  prescriptions: {p1->m1,m3; p3->m2}
 ->remove_interaction("m2","m4")
  state 18
  patients:      {p1,p2,p3}
  medications:   {m1,m2,m3,m4}
  interactions:  {m1->m2,m2->m1}
  prescriptions: {p1->m1,m3; p3->m2}
 ->add_prescription("p3","m4")
  state 19
  patients:      {p1,p2,p3}
  medications:   {m1,m2,m3,m4}
  interactions:  {m1->m2,m2->m1}
  prescriptions: {p1->m1,m3; p3->m2,m4}
```

TABLE XII
ERROR MESSAGES IN USE CASE

```
 ->add_prescription("p3","m4")
  state 17 Error e4: this prescription dangerous
  patients:      {p1,p2,p3}
  medications:   {m1,m2,m3,m4}
  interactions:  {m1->m2,m2->m1,m2->m4,m4->m2}
  prescriptions: {p1->m1,m3; p3->m2}
 ->remove_interaction("m2","m4")
  state 18
  patients:      {p1,p2,p3}
  medications:   {m1,m2,m3,m4}
  interactions:  {m1->m2,m2->m1}
  prescriptions: {p1->m1,m3; p3->m2}
 ->add_prescription("p3","m4")
  state 19
  patients:      {p1,p2,p3}
  medications:   {m1,m2,m3,m4}
  interactions:  {m1->m2,m2->m1}
  prescriptions: {p1->m1,m3; p3->m2,m4}
```

is signalled.
- In the generated code as shown in Table XIII, there are two clusters that the software designer must develop: `user_commands` and `model`. An example of a user command is shown in the Table.
- The main work for the software engineer is to develop the business logic in the model cluster.
- As shown in the UML diagram in Table XIII, the user interface is decoupled from the business logic (i.e. the model). The business logic may easily be interfaced with different concrete user interfaces (e.g. a web application or desktop application).
- Once the business logic is developed, it can be verified by executing the use cases as part of acceptance testing. These tests will also verify that the various classes in the business logic satisfy their specifications (preconditions, postconditions, class invariants etc.).
- Mathmodels may be used to specify the business logic (see sequel).

The Eiffel Testing Framework (ETF) thus helps software engineers to write and execute high-level, input-output-based use cases that are also acceptance tests. Inputs are specified as traces of user inputs (operations). Outputs (from executing each operation in the input trace) are by default logged onto the terminal, and their formats may be customized. The boundary of the system under development is defined by declaring the list of input operations (and their parameters) that might occur.

V. ETF Requirements Validation

In the previous section, we used ETF to build a framework for our EHealth system, generating an abstract user interface that can be tested with use cases. We now need to design the business logic (in the model cluster). The model must be capable of storing the medical data and updating it while preserving the consistency of the data. For example, requirement REQ6 (section IV-B) requires that “No prescription may have a dangerous interaction”. This is a safety critical requirement.

For this we need to specify the abstract state (already encoded in the use case in Table XI). There are sets of patients and medications, dangerous interaction relations, and patient prescriptions, and error messages. We would like build a model of the business logic and prove that that it satisfies the consistency and safety constraints. For this we need a formal model that can be used to predict the behaviour of the design.

Model building is not the same as programming. A model of a software product describes the properties that the program must fulfil. It does not describe the implementation data structures and algorithms contained in the program but rather the way by which we can eventually judge that the final program is correct. For example, a model of a file sorting program does not explain how to sort. Rather, it describes what the properties of a sorted file are and which relationship exists between the non-sorted file and the final sorted one. Abstract specifications are easier to understand and validate than detailed ones. They are also more easily analyzed using tools such as model checkers or theorem provers. In our case study, we abstract from many details. What exactly can and should be left out in a high-level model is of course specific to the problem; it also requires judgment that only experience can teach.

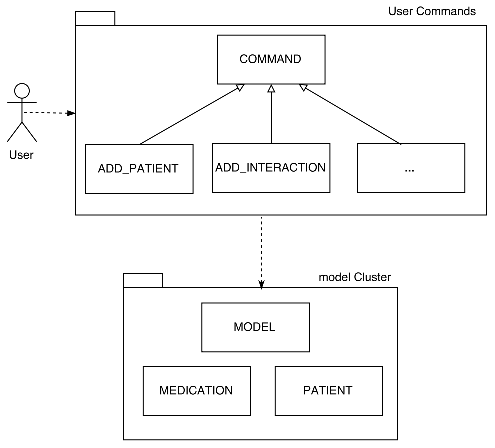
Modelling has to be accompanied by reasoning. The model of a program is not just a piece of text, whatever the formalism

TABLE XIII
Generated Code to be edited by software developers

Generated Eiffel files (`*.e`) for user commands and the model

```
|--user_commands
  |--add_interaction.e
  |--add_medication.e
  |--add_patient.e
  |--add_prescription.e
  |--command.e
  |--remove_interaction.e
  |__remove_prescription.e
|--model
  |--model.e
  |--health_system.e
  |--interaction.e
  |--medication.e
  |__patient.e
```

Class Diagram showing that the user cluster depends on the model



As an example, the generated Eiffel files `etf_add_patient.e` looks as follows:

```
class ADD_PATIENT inherit COMMAND feature
  add_patient(p: STRING)
    local
      l_p: PATIENT
    do
      -- create a patient from the model cluster
      create l_p.make (p)
      if model.patients.has(l_p) then -- error
        model.set_error ("e1: patient already entered")
      else -- update the model by adding the patient
        model.add_patient (l_p)
      end
      -- execute the command in interactive mode
      -- or store it for batch mode
      command.on_change.notify (Current)
    end
```

Initially the body of user input class ADD_PATIENT contains only the command update. The model cluster contains the business logic and model classes such as PATIENT, MEDICATION etc. are developed by the software developers.

being used. It also contains proofs that are related to this text that allow us to predict the behaviour of the final product. We must justify what we write by proving some consistency and safety properties. Software practitioners are not used to constructing such proofs, whereas people in other engineering disciplines are far more familiar with doing so. And one of the difficulties to make this become part of the daily practice of software engineers is the lack of good proving tool support for proofs, which could be used on a large scale.

Two main approaches to the formal verification are based, respectively, on model checking (algorithmic verification [25], [26]) and theorem proving (deductive verification). These two approaches have complementary strengths and weaknesses. Theorem proving is not limited to bounded domains but is not automatically decidable and often requires expertise and manual intervention to discharge the proofs. Model checking suffers from the state explosion problem and requires that we bound the domain. However, once the model and properties to be checked are satisfied, the check itself is automatic. If a model does not satisfy its required properties a useful counterexample is generated [27].

TLA+ is formal specification language for describing and reasoning about distributed and concurrent systems based on mathematical logic and set theory and linear time temporal logic [16], [28].

TLC is an on-the-fly model checker for debugging TLA+ specifications. Other model checkers require specifications to be written in primitive, low-level languages. TLC handels a subclass of TLA+ specifications that seems to include the ones that arise in describing actual systems at the mathematical level. Explicit-state model checkers like TLC must generate all reachable states. Real system specifications are usually not finite state, as they may contain unspecified sets of processors and unbounded message queues. TLC is invoked on the actual specification, using a separate configuration file that specifies a finite-state instance. TLC can also be used to generate finite-length random simulations of even infinite-state specifications.

The TLA Toolbox is an IDE (integrated development environment) for the TLA+ tools. It may be used to create and edit models, view pretty-printed versions of modules and to run the TLC model checker. The Toolbox provides an error trace viewer.

The TLA Toolbox is used to build abstract state machines with similarities to the description of the Mathmodels library in section II. In the case of Mathmodels, specifications are checked at runtime. In the case of TLA, we will use the model checker to exhaustively check a bounded domain (for example for a bounded set of patients, medications, interactions and prescriptions). This will demonstrate that the model of the business logic is feasible, and preserves the relevant consistency and safety constraints.

In the next section, we show that it is relatively seamless to refine the TLA model into a Mathmodels specification so that we can produce an implemented software product. Below, we use the TLC model checker to validate our EHealth specification.

The TLC model is likely to be an abstraction for part of the complete model. For example, the description of the EHealth system in Table IX is beyond the modelling capabilities of TLC. Type VALUE is an arbitrary precision real value (used for medication prescription amounts) which is not handled by the model checker. Also, the number of states specified by the complete system is likely to be beyond the analysis capability of the model checker due to combinatorial explosion of states.

A. Validating safety properties of EHealth with TLC

Fig. 1 and Fig. 2 provide a TLA specification of the EHealth system. In Fig. 1, the variables are *prescriptions* and *interactions*. TLA is untyped and thus we define a *TypeOK* predicate to check that these are relations of the relevant type in any execution of the system. The predicate following predicates are also defined:

- *Irreflexive* encodes the environmental description ENV4 in section IV-B;
- *Symmetry* encodes ENV3;
- *SafePrescription* encodes the safety critical property REQ6 asserting that no prescription shall have a dangerous interaction.

By providing numbered atomic requirements in section IV-B, we are thus able to trace where the informal requirements of enter into the mathematical model.

In Fig 2 we show the specifications of some of the actions such as adding an interaction or adding a prescription. Our TLA specification has all the user commands defined in the abstract ETF grammar of section IV-C.

The IF part in each action (user input in the grammar) models the precondition that must hold in the business logic for the operation to succeed. If the precondition does not hold, then an error message is provided to the user.

Each action is a two state predicate, i.e. a predicate that asserts what holds in the pre-state and what must hold in the post-state. For example, suppose we have three variables x, y and z and we wish to describe an action that increments x and sets y to x, provided $x + y \geq 0$ (the precondition), then the two-state formula describing the action is written:

$$x + y \geq 0 \ \wedge \ x' = x + 1 \ \wedge \ y' = x \wedge z' = z$$

where the primed versions are the values of the variables in the post-state. In Eiffel, we would write x = **old** $x + 1$ rather than $x' = x + 1$. So there is a semantic gap between these two specification notations, but the gap is not that great. In the next section, we encode these actions using the Mathmodels library, thus further refining requirements into a design.

The *Next* predicate in Fig 2 is defined as the disjunction of the various actions (user inputs). It defines the next-state relation of the EHealth system. The *Next* predicate asserts that any one of the actions may be taken. *Init* denotes the initial condition. The overall specification of the system is given by the temporal formula *Spec*, which is the conjunction $Init \wedge \Box[Next]_{vars} \wedge Live$, where *Live* might be some fairness constraints.

Spec thus specifies a transition system (or state machine) with fairness constraints. The temporal logic formula $\Box[Next]_{vars}$ specifies that every transition either satisfies the

Fig. 1. TLA Specification of EHealth

```
MODULE $ehealth3$

EXTENDS $Naturals, TLC$
CONSTANTS
      $M$   set of medications
    , $P$   set of patients
VARIABLES
      $prescriptions$
    , $interactions$
$vars \triangleq \langle prescriptions, interactions \rangle$
$TypeOK \triangleq$
    $\land$ $prescriptions \subseteq P \times M$
    dangerous interactions
    $\land$ $interactions \subseteq M \times M$
$Init \triangleq$
    $\land$ $prescriptions = \{\}$
    $\land$ $interactions = \{\}$
Invariants

$Symmetry \triangleq \forall m1, m2 \in M :$
  $\langle m1, m2 \rangle \in interactions \equiv$
  $\langle m2, m1 \rangle \in interactions$

A medication does not interact with itself
$Irreflexive \triangleq \forall m \in M :$
  $\langle m, m \rangle \notin interactions$

All prescriptions are safe
$i.e.$ do not have dangerous interactions
$SafePrescriptions \triangleq \forall m1, m2 \in M, p \in P :$
    $\langle m1, m2 \rangle \in interactions \Rightarrow$
      $\neg(\langle p, m1 \rangle \in prescriptions$
    $\land \langle p, m2 \rangle \in prescriptions)$
```

Fig. 2. TLA Specification of EHealth (cont.

```
Some of the Actions
. . .
$add\_interaction(m1, m2) \triangleq$
  IF
    $\land$ $m1 \in M$
    $\land$ $m2 \in M$
    $\land$ $m1 \neq m2$
    $\land$ $\langle m1, m2 \rangle \notin interactions$
    $\land$ $\forall p \in P : \neg(\langle p, m1 \rangle \in prescriptions$
                      $\land \langle p, m2 \rangle \in prescriptions)$
  THEN
    $\land$ $interactions' =$
        $interactions \cup \{\langle m1, m2 \rangle, \langle m2, m1 \rangle\}$
    $\land$ UNCHANGED $prescriptions$
  ELSE
    $\land$ UNCHANGED $\langle prescriptions, interactions \rangle$
    $\land$ $Print(\langle$"add_interaction", "e1"$\rangle$, TRUE)

$add\_prescription(p, m) \triangleq$
  IF
    $\land$ $p \in P$
    $\land$ $m \in M$
    $\land$ $\langle p, m \rangle \notin prescriptions$
    $\land$ $\forall x \in M : \langle p, x \rangle \in prescriptions \Rightarrow$
        $\langle x, m \rangle \notin interactions$
  THEN
    $\land$ $prescriptions' = prescriptions \cup \{\langle p, m \rangle\}$
    $\land$ UNCHANGED $interactions$
  ELSE
    $\land$ UNCHANGED $\langle prescriptions, interactions \rangle$
    $\land$ $Print(\langle$"add_interaction", "e2"$\rangle$, TRUE)

$Next \triangleq$
    $\lor \exists m1, m2 \in M : add\_interaction(m1, m2)$
    $\lor \exists m \in M, p \in P : add\_prescription(p, m)$
. . .

$Spec \triangleq Init \land \Box[Next]_{vars}$
```

action formula *Next* or leaves the expression $vars$ unchanged. This admits "stuttering transitions" that do not affect the variables of interest. Stuttering invariance is a key concept of TLA that simplifies the representation of refinement, as well as compositional reasoning. The initial condition and the next-state relation specify how the system may behave. Fairness conditions complement this by asserting what actions must occur (eventually).

The TLC model checker is then invoked and is used to prove that in a bounded domain of patients and medications the type correctness properties and safety properties hold in all possible states of the transition system. The model checker can also prove that the bounded system does not deadlock and can prove other liveness properties.

The TLA specification can now be used to design the business logic in Eiffel.

VI. ETF, Mathmodels and Verification of the Design

In the previous section, we developed a TLA specification for the EHealth system and validated the specification via model checking. The actions in the specification were themselves derived from the ETF grammar of user inputs in section IV-C, and the description of the abstract state in use cases (such as Table XI).

An important part of the validation was the derivation and proof that the system safety invariants are preserved by the specification. Such system invariants are invaluable for ensuring the safety of mission critical systems. The specification language that TLC can check is a subset of the TLA+ language. Each construct in this untyped subset has an analogue in the Mathmodels typed language, although the

TABLE XIV
PARTS OF CLASSES FOR MEDICATION AND INTERACTION

```
class MEDICATION feature ...
  name: STRING

  maps_to alias ‘‘|−>’’ (other: MEDICATION): INTERACTION
        −− maps to operator Current ↦ other
        do
           create Result.make (Current, other)
        end
end

class INTERACTION feature ...
   first: MEDICATION
   second: MEDICATION
end
```

TABLE XV
SOME QUERIES OF MATHMODELS REL[G, H]

```
class REL [G, H] inherit
   SET[TUPLE[G, H]]
create
   make_empty
feature −− queries
   domain: SET [G]
      −− Return the domain set of relation.
   range: SET [H]
      −− Return the range set of relation.
   image alias [] (g: G): SET [H]
      −− Retrieve set of range items
    −− for domain element g
    extended alias ”+” (p:TUPLE[G, H]): REL [G, H]
         −− return a new relation with addition of t
    overriden_by (p: TUPLE[G, H]): REL [G, H]
      −− Return a new relation the same as Current,
      −− except p.first now maps to p.second
      −− alias ‘‘@<+’’

    ...
end
```

Mathmodel constructs are more verbose. Thus, the semantic gap in going from TLC to a Mathmodels specification is relatively small as TLC and Mathmodels both specify via mathematical descriptions using sets, sequences, functions, relations and bags.

In this section, we refine the TLA specification into a Mathmodels specification of the business logic, as Eiffel program text containing specification and an implementation. The program text contain specifications as contracts and an implementation so as to obtain an executable system. The use cases developed in the requirements task can be re-used as acceptance tests of the final software product. When these acceptance tests execute, the implementation is also verified to satisfy the specification.

We develop the business logic in the model cluster of the generated code as shwon in Table XIII where we add classes such as PATIENT, INTERACTION and MEDICATION (see Table XIV).

The actions (user inputs) are specified in class *HEALTH_SYSTEM* in Table XVI where we define the abstract state variables such as:

- *prescriptions*, which is a set of interactions with type SET[INTERACTION];
- *interactions* which is a relation between patients and medications with type REL[PATIENT, MEDICATION]. Class *REL*[*G*, *H*] is part of Mathmodels and is shown in Table XV.

A. Important system safety invariants

In the TLA specification in the previous section, we identified some important system invariants such as symmetry, irreflexivity and the safety invariant asserting that patients are never prescribed dangerous interactions. These invariants themselves came from the numbered atomic requirements during the elicitation phase described in section IV-B. These invariants now become part of the program text in class *HEALTH_SYSTEM* in Table XVI:

- Atomic requirement ENV3 (symmetry) is shown at lines 51 to 56;
- Atomic requirement ENV4 (irreflexivity) is shown at lines 57 to 60;
- The safety requirement REQ6 is shown at lines 61 to 72.

In each case, the invariants in the Eiffel program text (using Mathmodels) is close to the TLA specifications.

In order to preserve each of these crucial invariants, the actions (user inputs) must have preconditions that are guaranteed to ensure the invariants.

Some of these actions are shown in the program text in Table XVI. For example, consider the precondition of command *add_prescription*(*p*,*m*) in *HEALTH_SYSTEM* starting at line 31:

- Line 35 asserts that patient p must be in the system and line 37 asserts that medication m is not yet prescribed for patient p. The query *prescription*[*p*] (from class REL) is the relational image returning a set of medications for p.
- Lines 40 to 42 assert that adding this medication does not create a dangerous interaction. This part of the precondition ensures that the system safety invariant REQ6 is preserved. The query *prescriptions*[*p*] is the relational image for patient *p* and returns all the medications in the range prescribed for *p*.

Commands in the *HEALTH_SYTEM* in the model cluster of the generated code (such as *add_prescription*(*p*,*m*)) are given demanding preconditions whereas the analogous command routines in the user interface cluster (see Table XIII) have no preconditions and apply defensive programming. Why is this?

The software designer has no control over the users that provide data at the user interface. Thus, the command *add_prescription*(*p*,*m*) at the user interface has to deal with good inputs as well as possible erroneous inputs. Thus there cannot be a precondition at the user interface. If the input data is legal, then the user interface can invoke the command in the business logic (i.e. in the model cluster). If it is not legal it must signal to the external users that the inputs are problematic.

Each of the contracts in class *HEALTH_SYSTEM* (the business logic) holds between a routine (the supplier) and another routine (its caller at the user interface): we are concerned about software-to-software communication, not software-to-human or software-to-outside-world. A precondition is not used to take care of correcting user input. It would be wishful thinking, not a reliability technique, to have demanding preconditions at the user interface to the external world.

At the user interface, there is no substitute for the usual condition-checking constructs for input validation. Any inputs from the outside world including input data and sensor data in a real-time system needs that kind of checking. In obtaining information from the outside one cannot rely on preconditions. Thus there is no precondition at the user interface to the external world. The task of the input modules at the interface to the external world is to guarantee that no information is passed to the business logic, as that would cause inconsistent data.

B. Use Cases, DbC and Acceptance Testing

Dbc views a software system as a set of components whose collaboration is based on precisely defined specifications of mutual obligations—the contracts. The central idea of this method is to inherently embed the contracts in the code and validate them automatically at run time. Doing so consistently has two major benefits: (a) It automatically helps detect bugs (as opposed to "handling" them), and (b) it is one of the best ways to document code.

As mentioned earlier, the use cases derived during requirements elicitation can also be used to do acceptance testing on the implemented code. As the acceptance tests are executed, the contracts are exercised and we thus verify the implementation against the specification.

VII. Comparison with other approaches

There is a significant body of work on models, contracting mechanisms and their analysis. The paper [29] provides a survey of contracting mechanisms, comparing Eiffel with other frameworks developed for Java and C# for object-oriented specification and verification.

Most interface specification languages use some variation on Hoare's pre- and postcondition technique in languages such as Z, VDM and the Larch family of interface specification languages, using a specialized mathematical vocabulary. Specifications operate on abstract values, which are abstractions of the "concrete" state of the program. The operations used on abstract values are mathematical, and thus an excellent fit for formal manipulation with theorem provers.

Experience with Larch-style interface specification languages indicates that a mathematical syntax for assertions which is different than the programming language's syntax, is a barrier to use by programmers. Programmers seem more comfortable with an assertion language that is based on the programming language's own expression syntax. This is the approach followed by Gypsy, Anna, APP, and Eiffel, and adopted by JML and Spec#. OpenJML is a recent version of the notations and ideas behind JML.[8]

Several of the challenges in the design of such Eiffel-like interface specification languages such as JML and Spec# stem from this fundamental decision to write assertions using a subset of the expressions in the underlying programming language. One of the basic problems is to overcome the mismatch between the programming language's expressions and the needs of automatic theorem provers and model checkers. It is essential that verification techniques are modular, that is, that they allow one to reason about a class independently of its clients and subclasses.

Modularity is crucial to verify reusable classes such as library classes and for scalability. Many of the challenges stem from this modularity requirement. They call for modular solutions to problems for which non-modular solutions already exist. The specification and verification challenges described in [29] are challenges for specification and verification methods, i.e. how to apply existing concepts, formalisms and logics to specify and verify a program.

The paper in [30] presents an integrated development environment for Dafny—a programming language, verifier, and proof assistant—that addresses issues present in most state-of-the-art verifiers: low responsiveness and lack of support for understanding non-obvious verification failures. The paper demonstrates several new features that move the state-of-the-art closer towards a verification environment that can provide verification feedback as the user types and can present more helpful information about the program or failed verifications in a demand-driven and unobtrusive way.

The most pressing problem in Dafny is what to do with verification tasks that require a long time. When a method is long and difficult, it has to be manually broken up into smaller pieces. Time-outs occur in some part of any larger proof attempt, especially those that involve large recursive functions or non-linear arithmetic, while the user is working on getting the verification through. Currently, the verifier does not produce as much information for verification attempts that time out as it does for attempts that fail.

SPARK Pro is an integrated static analysis toolsuite for verifying high-integrity software through formal methods [31]. It supports the SPARK 2014 language and provides advanced verification tools that are tightly integrated into the GNAT Programming Studio. Using SPARK Pro, developers can formally define and semi-automatically verify software architectural properties, and guarantee a wide range of software integrity properties such as freedom from run-time errors, enforcement of security policies, and functional correctness (compliance with a formally defined specification). This automated verification is particularly well-suited to applications where software failure is unacceptable.

Systems such as those described above using theorem provers have been used on up to about 30,000 lines of code. An advantage is that if the verification succeeds, then we have a proof of correctness, that transcends what testing can

[8]http://www.openjml.org.

do. However, manual intervention is sometimes required and expertise is needed.

By contrast, in runtime checking such as in Eiffel, proofs are lacking but very large systems can be handled for verification. Manual intervention is not needed as it is in theorem proving.

The TLC model checker used in this paper for the analysis of specifications has been used in industry such as at Amazon [32]. Amazon builds many sophisticated distributed systems that store and process data on behalf of their customers. In order to safeguard that data they rely on the correctness of an ever-growing set of algorithms for replication, consistency, concurrency-control, fault tolerance, auto-scaling, and other coordination activities. Achieving correctness in these areas is a major engineering challenge as these algorithms interact in complex ways in order to achieve high-availability on cost-efficient infrastructure whilst also coping with relentless rapid business-growth.

Amazon has used TLA+ on 10 large complex real-world systems. In every case TLA+ has added significant value, either preventing subtle serious bugs from reaching production, or giving enough understanding and confidence to make aggressive performance optimizations without sacrificing correctness. Executive management are now proactively encouraging teams to write TLA+ specifications for new features and other significant design changes. Model checking rather than theorem proving has been a major ingredient of this success. The Amazon report mentions the following:

- *Get design right*. Formal methods help engineers get the design right, which is a necessary first step toward getting the code right. If the design is broken, then the code is almost certainly broken, as mistakes during coding are extremely unlikely to compensate for mistakes in design. Engineers are likely to be deceived into believing the code is correct because it appears to correctly implement the (broken) design.
- *Gain better understanding*. Formal methods help engineers gain a better understanding of the design. Improved understanding can only increase the chances they will get the code right.
- *Write better code*. Formal methods can help engineers write better "self-diagnosing code" in the form of assertions. Experience suggest pervasive use of assertions is a good way to reduce errors in code. An assertion checks a small, local part of an overall system invariant. A good system invariant captures the fundamental reason the system works; the system will not do anything wrong that could violate a safety property as long as it continuously maintains the system invariant. The challenge is to find a good system invariant, one strong enough to ensure no safety properties are violated. Formal methods help engineers and strong invariants, so formal methods can help improve assertions that help improve the quality of code.

So far, the benefits that Amazon have gained from formal methods have arisen from writing precise specifications to eliminate ambiguity, and model-checking finite models of those specifications to try to find errors with the TLC model checker [33]. Amazon has already run into the practical limits of model-checking, caused by combinatorial state-explosion. In one case, they found a serious defect that was only revealed in an execution trace comprising 35 steps of a high-level abstraction of a complex system. Finding that defect took several weeks of continuous model-checking time on a cluster of 10 high-end machines, using carefully crafted constraints to bound the state-space. Even when using such constraints the model-checker still had to explore billions of states.

The Amazon engineers write [33]: "In industry, engineers are extremely skeptical of proofs. Engineers strongly doubt that proofs can scale to the complexity of real-world systems, so any viable proof method would need an effective mechanism to manage that complexity. Also, most proofs are so intricate that there is more chance of an error in the proof than an error in the algorithm, so for engineers to have confidence that a proof is correct, we would need machine verification of the proof. However, most systems that we know of for machine-checked proof are designed for proving conventional theorems in mathematics, not correctness of large computing systems. TLA+ has a proof system that addresses these problems. The TLA+ proof system (TLAPS) uses structured hierarchical proof, which we have found to be an effective method for managing very complex proofs. TLAPS works directly with the original TLA+ specification, which allows users to first eliminate errors using the model checker and then switch to proof if even more confidence is required. ... We have tried TLAPS on small problems and found that it works well. However, we have not yet proved anything useful about a real system." My understanding is that TLAPS does not extend to liveness proofs.

Microsoft is currently designing an advanced new database called Cosmos DB. TLA+ is being used to specify some of the complex parts so that they can be checked for correctness.[9]

The report [33] compares TLA+ with Alloy, VCC, Event-B, PVS, Coq. We also refer the reader to [2] for a comparison between the various specification formalisms and tools.

Software testing is an important activity in the software development life-cycle. The report [34] surveys various forms of testing. A survey of practitioners in Canada reveals: (1) the importance of testing-related training is increasing; (2) functional and unit testing are two common test types that receive the most attention and efforts spent on them; (3) usage of the mutation testing approach is getting attention; (4) traditional Test-last Development (TLD) style still dominates and a few companies are attempting the new development approaches such as Test-Driven Development (TDD); and Behavior-Driven Development (BDD); (5) in terms of the most popular test tools, NUnit and Web application testing tools overtook JUnit and IBM Rational tools; (6) most Canadian companies use a combination of two coverage metrics: decision (branch) and condition coverage; (7) the number of passing user acceptance tests and number of defects found per day (week or month) are regarded as the most important quality assurance metrics and decision factors to release; (8)

[9]http://techcrunch.com/2017/05/10/with-cosmos-db-microsoft-wants-to-build-one-database-to-rule-them-all/, accessed 31 July, 2017.

testers are out-numbered by developers. Various forms of acceptance testing are reported in [35].

The report [36] describes how the activities involved in testing software are known to be difficult and time consuming. Among them is the isolation of faults once failures have been detected. The paper investigate how the instrumentation of contracts addresses this issue. Contracts are known to be a useful technique to specify the precondition and postcondition of operations and class invariants, thus making the definition of object-oriented analysis or design elements more precise. The authors conclude that instrumented contracts are valuable and reduce the cost of testing.

In [37], Design by contract (DbC) is analyzed in the context of Java. The authors write that there exists ample support for DbC for sequential programs. Applying DbC to concurrent programs presents several challenges. Using Java as the target programming language, the authors tackle such challenges by augmenting the Java Modelling Language (JML) and modifying the JML compiler to generate runtime assertion checking code to support DbC in concurrent programs. They apply their solution to a carefully designed case study on a highly concurrent industrial software system from the telecommunications domain to assess the effectiveness of contracts as test oracles in detecting and diagnosing functional faults in concurrent software. Based on these results, The main results include that contracts of a realistic level of completeness and complexity can detect around 76 percent of faults and reduce the diagnosis effort for such faults tenfold.

In Eiffel, concurrency and contracts have been implemented using the SCOOP notation and mechanism [38], [39]. ETF and Mathmodels are compatible with SCOOP concurrency.

The report [40] introduces a Mathematical Model Library (MML) which is a precursor to Mathmodels. The author reuses the capabilities of the Eiffel programming language to express mathematical expressions. All mathematical operations are immutable yielding new values that do not change the existing ones. Model classes may not have commands. Queries in a model class may only rely on queries of the class itself and public queries of other model classes. Model objects are never compared by reference. At about the same time, this author and his students [41] also used model libraries in Eiffel together with a theorem prover for proving properties.

The authors of [42] present their experience verifying the full functional correctness of an Eiffel-Base2 container library offering all the features customary in modern language frameworks, such as external iterators, and hash tables with generic mutable keys and load balancing. Verification uses the automated deductive verifier AutoProof. The results indicate that verification of a realistic container library (135 public methods, 8,400 LOC) is possible with moderate annotation overhead (1.4 lines of specification per LOC) and good performance (0.2 seconds per method on average).

The Mathmodels container library differs from MML and Eiffel-Base2 in many ways. Mathmodels uses runtime verification rather than theorem proving for scalability to very large systems because the checking is completely automatic. Also, Mathmodels is Void safe [43] whereas the others are not. For proofs, we use the TLC model checker for the complex parts that are harder to get right. This is based on our judgement that engineers (as can be seen in the Amazon study) are prepared to learn specification languages provided the subsequent analysis can be done automatically. Finally, Mathmodels has an immutable mathematical part (the queries) as well as mutable commands. Both are useful as demonstrated in the section on abstract specification of the Quicksort routine and the EHealth example.

The ETF Tool described in this paper does not seem to have an analogue in the literature. It is used at the requirements elicitation phase for use cases as well as for acceptance testing after implementation. Its support for many software engineering tasks, the specification of an abstract user interface decoupled from the business logic, the generation of appropriate code to facilitate the design based on developing an abstract state machine make it useful in the development of reliable software. It is implemented in Eiffel, but the basic idea may be reused in any modern language or framework.

VIII. Conclusion

In this paper we described two tools—ETF and Mathmodels—for developing reliable software by eliciting precise software requirements, validating them and verifying that the final software product satisfies the requirements. Mathmodels extends the classical Eiffel contracting notation with the use of mathematical models (sets, sequences, relations, functions, bags) to describe abstract state machines. Classical contracts are incomplete or are low level implementation assertions. Mathmodel contracts provide complete specifications of components and systems that can be verified via runtime contract checking that scales up to verifying very large systems. Mathmodels are void safe and have immutable queries (for specifications) as well as relatively efficient mutable commands for the abstract description of algorithms.

The ETF tool is used in requirements elicitation to derive specifications, to describe the user interface, to identify the abstract state, and to develop use cases before the software product is constructed. The ETF tool generates code that decouples the user interface from the design (the business logic). The same use cases are used as acceptance tests when the final product is completed. The ETF Tool supports the derivation of important system safety invariants which become Mathmodel class invariants in the production code.

The ideas can be extended to other languages and frameworks and are placed in the context of best practices for software engineering. Important safety invariants are traceable all the way from requirements elicitation and analysis, to design and the construction of final code.

Reliable systems must be correct and robust. Robustness is the degree to which a system continues to function in the presence of invalid inputs or stressful environmental conditions. It is the ability of software systems to react appropriately to abnormal conditions. ETF provides a method to ensure the production code does not fail with crashes or exceptions to invalid inputs but provides appropriate responses while protecting the integrity of the business logic and data.

Software security is also important in the production of reliable software. One of the most visible signs of this phenomenon is when Microsoft halted further development (in 2001) in favour of code reviews for hunting down security flaws. This paper does not directly deal with security issues. But, many security flaws such as buffer overflows are the result of poor software engineering practices. Improving security implies taking a coherent look at best software engineering practices and tools.

In [44], Parnas et. al. describes how many have sought a software design process that allows a program to be derived systematically from a precise statement of requirements. The authors proposes that, although designing a real product in that way will not usually be successful, it is possible to produce documentation that makes it appear that the software was designed by such a process. The ideal process and the documentation that it requires are described in the paper. The authors explain why one should attempt to design according to the ideal process and why one should produce the documentation that would have been produced by that process, and the contents of each of the required documents are outlined. In this paper, we have proposed a variety of important documents including E/R-descriptions (in section IV), and the contract model specifications and system invariants in the production text (see section VI). The advantage of the latter is that the contracting mechanism ensures that the specifications are kept in sync with the implementations.

A. Computer Science and Software Engineering Education

I have used the Mathmodels and ETF Tools in a third year software design course with students from computer science, software engineering and computer engineering. In the course, we teach conventional topics such as design patterns, information hiding, modularity, testing and good documentation practice. But we also teach the value of contracting and the importance of system invariants. Students have mentioned that they learn most from the design project. The ETF Tool allows us to provide students with testable specifications free of design and implementation detail, where the user interface is decoupled from the design. Thus the students must do a design from scratch, implement it and document it, but we can also test their design correctness via a comprehensive set of use cases provided as part of the specification.

Thomas Ball is a principal researcher and co-manager of the Research in Software Engineering (RiSE) group at Microsoft Research and Benjamin Zorn is a principal researcher and co-manager of the Research in Software Engineering (RiSE) group at Microsoft Research. In an article titled "Teach Foundational Language Principles: Industry is ready and waiting for more graduates educated in the principles of programming languages", they make some recommendations for computer science education looking to the future.

They write that experiences with bugs like the recent TLS heartbeat buffer read overrun in OpenSSL (Heartbleed) show the cost to companies and society of building fundamental infrastructure in dated programming languages with weak type systems (the C language in this case) that do not protect their abstractions. The suggestion is that students be taught some of the new specification languages, which allow the designers of systems and algorithms to gain more confidence in their design before encoding them in programs where it is more difficult to find and fix design mistakes. Recently, Pamela Zave of AT&T Labs showed the protocol underlying the Chord distributed hash table is flawed by modelling the protocol in the Alloy language. Emina Torlak and colleagues used a similar modelling approach to analyze various specifications of the Java Memory Model (JMM) against their published test cases, revealing numerous inconsistencies among the specifications and the results of the test cases [45].

Ball and Zorn write: "Our recommendations are threefold, visiting the three topics discussed in this Viewpoint in reverse order (formal design languages, domain-specific languages, and new general-purpose programming languages). First, computer science majors, many of whom will be the designers and implementers of next-generation systems, should get a grounding in logic, its application in design formalisms, and experience the creation and debugging of formal specifications with automated tools such as Alloy or TLA+. As Leslie Lamport says, 'To designers of complex systems, the need for formal specs should be as obvious as the need for blueprints of a skyscraper.' The methods, tools, and materials for educating students about 'formal specs' are ready for prime time. Mechanisms such as 'design by contract,' now available in mainstream programming languages, should be taught as part of introductory programming, as is done in the introductory programming language sequence at Carnegie Mellon University. Students who learn the benefits of principled thinking and see the value of the related tools will retain these lessons throughout their careers. We are failing our computer science majors if we do not teach them about the value of formal specifications."

Acknowledgment

The author would like to thank Chen-Wei (Jackie) Wang and Simon Hudon who are significant contributors to ETF and Mathmodels.

References

[1] L. Lamport, "Who builds a house without drawing blueprints?" *Communications of the ACM*, vol. 58, no. 4, 2015.

[2] H. Habrias and M. Frappier, Eds., *Software Specification Methods*. ISTE, 2006.

[3] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *2007 Future of Software Engineering*, ser. FOSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 37–54. [Online]. Available: http://dx.doi.org/10.1109/FOSE.2007.14

[4] R. F. Paige, J. S. Ostroff, and P. Brooke, "Principles of Modeling Language design," *Information and Software Technology*, vol. 42, pp. 665–675, 2000, principles of Modeling Language design.

[5] R. F. Paige, P. J. Brooke, and J. S. Ostroff, "Metamodel-based model conformance and multi-view consistency checking," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 3, 2007.

[6] MBE-Subcommittee, "Final report of the model based engineering (MBE) subcommittee," National Defense Industrial Assoctaion (NDIA) Systems Engineering DivisionNDIA) Systems Engineering Division, Tech. Rep., 2011.

[7] H. Gronninger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel, "Textbased modeling," in *Proceedings of the 4th International Workshop on Software Language Engineering*, 2007.

[8] K. Walden and J.-M. Nerson, *Seamless Object Oriented Software and Architecture*. Prentice Hall, 1995, seamless Object Oriented Software and Architecture.

[9] B. Meyer, *Object-Oriented Software Construction*. Prentice Hall, 1997.

[10] *Touch of Class: Learning how to Program Well, with Objects and Contracts*. Springer Verlag, 2013.

[11] N. Polikarpova, C. A. Furia, Y. Pei, Y. Wei, and B. Meyer, "What good are strong specifications?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 262–271. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486823

[12] A. van Lamsweerde, *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009. [Online]. Available: http://eu.wiley.com/WileyCDA/WileyTitle/productCd-EHEP000863.html

[13] B. Meyer, *Agile! The Good, the Hype and the Ugly*. Springer, 2014.

[14] ——, *Dependable Software*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 1–33. [Online]. Available: https://doi.org/10.1007/11808107_1

[15] J. S. Ostroff, D. Makalsky, and R. Paige, "Agile specification driven development," in *Fifth International Conference on Extreme Programming and Agile Processes in Software Engineering XP2004*, Garmisch-Partenkirchen, Germany, 2004.

[16] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Pearson, 2002.

[17] C. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification and examples," *ACM Computing Surveys*, vol. September, 2014.

[18] L. Lamport, *The PlusCal Algorithm Language*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 36–60. [Online]. Available: https://doi.org/10.1007/978-3-642-03466-4_2

[19] D. Gries, *The Science of Programming*. Springer-Verlag, 1985.

[20] L. Lamport, "Teaching concurrency," *ACM SIGACT News*, vol. 40, no. 1, pp. 58–62, 2009.

[21] M. Jackson, *Software Requirements & Specifications: a lexicon of practice, principles and prejudices*. Addison-Wesley, 1995.

[22] FAA, "Requirements engineering management handbook," US Federal Aviation Administration, Tech. Rep. DOT/FAA/AR-08/32, 2009.

[23] J.-R. Abrial, *Modeling in Event-B*. Cambridge University Press, 2010.

[24] E. Hull, K. Jackson, and J. Dick, *Requirements Engineering*. SpringerVerlag, 2005.

[25] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "PAT: Towards Flexible Verification under Fairness," in *CAV*, ser. LNCS 5643, 2009, pp. 709 – 714.

[26] J. Ostroff, C.-W. Wang, S. Hudon, Y. Liu, and J. Sun, "TTM/PAT: Specifying and Verifying Timed Transition Models," in *FTSCS*, ser. Communications in Computer and Information Science. Springer, 2014, vol. 419, pp. 107–124.

[27] T. E. Uribe, *Combinations of Model Checking and Theorem Proving*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 151–170. [Online]. Available: https://doi.org/10.1007/10720084_11

[28] S. Merz, *The Specification Language TLA+*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 401–451. [Online]. Available: https://doi.org/10.1007/978-3-540-74107-7_8

[29] G. T. Leavens, K. R. M. Leino, and P. Müller, "Specification and verification challenges for sequential object-oriented programs," *Formal Aspects of Computing*, vol. 19, no. 2, pp. 159–189, Jun 2007. [Online]. Available: https://doi.org/10.1007/s00165-007-0026-7

[30] R. Leino and V. Wüstholz, "The dafny integrated development environment," in *Proceedings 1st Workshop on Formal Integrated Development Environment F-IDE*, April 2014. [Online]. Available: https://www.microsoft.com/en-us/research/publication/dafny-integrated-development-environment/

[31] C. Brandon and P. Chapin, *A SPARK/Ada CubeSat Control Program*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 51–64. [Online]. Available: https://doi.org/10.1007/978-3-642-38601-5_4

[32] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2699417

[33] C. Newcombe, *Why Amazon Chose TLA +*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 25–39. [Online]. Available: https://doi.org/10.1007/978-3-662-43652-3_3

[34] V. Garousi and J. Zhi, "A survey of software testing practices in canada," *Journal of Systems and Software*, vol. 86, no. 5, pp. 1354 – 1376, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121212003561

[35] G. Adzic, *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*. United Kingdom: Neuri Limited, 2009.

[36] L. C. Briand, Y. Labiche, and H. Sun, "Investigating the use of analysis contracts to support fault isolation in object oriented code," in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM Press, 2002, pp. 70–80.

[37] W. Araujo, L. C. Briand, and Y. Labiche, "On the effectiveness of contracts as test oracles in the detection and diagnosis of functional faults in concurrent object-oriented software," *IEEE Trans. Software Eng.*, vol. 40, no. 10, pp. 971–992, 2014. [Online]. Available: https://doi.org/10.1109/TSE.2014.2339829

[38] S. West, S. Nanz, and B. Meyer, "Efficient and reasonable object-oriented concurrency," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2015, San Francisco, CA, USA, February 7-11, 2015*, 2015, pp. 273–274. [Online]. Available: http://doi.acm.org/10.1145/2688500.2688545

[39] B. M. Piotr Nienaltowski1 and J. S. Ostroff, "Contracts for concurrency," *Formal Aspects of Computing*, vol. 21, no. 4, 2009.

[40] B. Schoeller, T. Widmer, and B. Meyer, "Making specifications complete through models," in *Architecting Systems with Trustworthy Components*, R. Reussner, J. Stafford, and C. Szyperski, Eds., vol. 3938. Springer-Verlag Lecture Notes in Computer Science, 2006.

[41] J. Ostroff, C.-W. Wang, E. Kerfoot, and F. A. Torshizi, "Automated model-based verification of object oriented code," in *Verified Software: Theories, Tools, Experiments (VSTTE Workshop, Floc 2006)*. Microsoft Research MSR-TR-2006-117, 2006.

[42] J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova, "Autoproof: Auto-active functional verification of object-oriented programs," in *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science. Springer, 2015.

[43] B. Meyer, "Ending null pointer crashes," *Commun. ACM*, vol. 60, no. 5, pp. 8–9, 2017. [Online]. Available: http://doi.acm.org/10.1145/3057284

[44] D. L. Parnas and P. C. Clements, "A Rational Design Process: How and Why to Fake it," *IEEE Trans. on Software Engineering*, vol. SE-12, no. 2, pp. 251–257, 1986, a Rational Design Process: How and Why to Fake it.

[45] T. Ball and B. Zorn, "Teach foundational language principles," *Commun. ACM*, vol. 58, no. 5, pp. 30–31, Apr. 2015. [Online]. Available: http://doi.acm.org/10.1145/2663342

Jonathan S. Ostroff Biography text here.

TABLE XVI
PARTS OF CLASS HEALTH_SYSTEM

```
class
  HEALTH_SYSTEM
feature −− queries ...
  patients: SET [PATIENT]
    −− set of patients
  medications: SET [MEDICATION]
    −− set of medications
  prescriptions: REL [PATIENT, MEDICATION]
    −− prescriptions
  interactions: SET [INTERACTION]
    −− dangerous interactions
feature −− commands
  add_interaction (m1, m2: MEDICATION)
      −− Add an interaction between ’m1’ and ’m2’.
    require
      medications.has (m1) and medications.has (m2)
      m1 ≠ m2
      not interactions.has (m1 ↦ m2)
      −− ∀p ∈ dom(prescriptions) : m1, m2 ⊈ prescriptions[p]
      across prescriptions.domain as pc all
        not (⟪ m1, m2⟫ ⊆ prescriptions[pc.item])
      end
    do
      interactions.extend ([m1, m2])
      interactions.extend ([m2, m1])
    ensure
        interactions ∼ old interactions + [m1, m2] + [m2, m1]
        −− UNCHANGED (patients, medications, prescriptions)
    end

  add_prescription (p: PATIENT; m: MEDICATION)
        −− Add a prescription of ’m1’ to ’p1’.
    require
      −−p ∈ patients
      patients.has (p)
      −− m ∉ prescriptions[p]
      not prescriptions[p].has (m)
      −− cannot cause a dangerous interaction
      −− ∀med ∈ prescriptions[p] : (med, m) ∉ interaction
      across prescriptions[p] as med all
        not interactions.has(med.item ↦ m)
      end
    do
      prescriptions.extend ([p, m])
    ensure
      prescriptions ∼ old prescriptions + [p, m]
      −− UNCHANGED (patients, medications, interactions)
  end
    ...
invariant
  symmetry_ENV3:
    across medications as m1 all
    across medications as m2 all
      interactions.has (m1.item ↦ m2.item)
      = interactions.has (m2.item ↦ m1.item)
    end end
  irreflexivity_ENV4:
    across medications as m1 all
      not interactions.has (m1.item ↦ m1.item)
    end
  no_dangerous_interactions_REQ6:
    across prescriptions.domain as p all
    across prescriptions[p.item] as m1 all
    across prescriptions[p.item] as m2 all
      interactions.has (m1.item ↦ m2.item) and m1.item ≁ m2.item
      implies
      not(prescriptions.has([p.item,m1.item])
      and prescriptions.has([p.item,m2.item]))
    end
    end
    end
  consistent_domain:
    prescriptions.domain ⊆ patients
end
```