Modelling and Testing Requirements via Executable Abstract State Machines

Jonathan S. Ostroff and Chen-Wei Wang
Department of Electrical Engineering and Computer Science
Lassonde School of Engineering, York University, Canada
Email: {jonathan, jackie}@eeecs.yorku.ca

Abstract—We describe a method and tools for deriving specification models from requirements, and for validating that the final software product satisfies the requirements. ETF (Eiffel Testing Framework) is a tool for generating code from an abstract grammar specification of user interface actions derived from the requirements document. Mathmodels extends the classical Eiffel contracting notation with the use of mathematical models (based on sets, sequences, relations, functions, bags). The Mathmodels library has immutable queries (for specifications) as well as relatively efficient mutable commands (for describing executable abstract state machines). Models are developed and validated using the industrial strength Eiffel IDE, and the use of these tools thus scale up to the development of large systems in a way that supports the derivation of specification models from requirements, and seamlessness between models and code.

Index Terms—requirements, models, specifications, validation, tools, reliable software

I. INTRODUCTION

Model Driven Engineering (MDE) holds the promise of raising the level of abstraction when designing systems by promoting domain specific modelling languages, model transformation techniques and code generation. The use of models above the code level is promoted as a method for handling the complexity of software development. Although there is a large body of research and industrial successes, there are still challenging issues especially with respect to keeping code in sync with the models [1]–[3].

The authors of [4] survey activities including the use of models for code generation, modelling language creation, and model-based testing, among a wide range of industrial users of model-driven engineering (see also [5]). They found it surprising that 35% of respondents do not use models in testing. The authors attribute this to the greater degree of formality and effort needed for the use of models for testing and simulation. Impedance mismatch between models and code continues to be a challenge [6].

An important facet of requirements engineering is to refine requirements into specifications [7]. Although requirements are often expressed in natural language [8], requirements models have also been used to elicit, document and analyze them. As described in [9], there is a tension between requirements models as a description of the problem space (entirely elaborated in terms of user and system environment) and specifications in the solution space (expressing at a high level of abstraction what the forthcoming artifact will do, with no concerns about how it will be done). As requirements engineering deals with both spaces, modelling techniques are used in both, even if certain techniques (e.g. UML) are more suited to the solution space, while other techniques (e.g. goal modelling) might be better suited for the problem space. In requirements engineering, MDE might be investigated as a technique to derive models in the solution space from models in the problem space.

The analysis of requirements models in [9] indicates a predominance of proposals for new languages for requirements representation. Also, there is significant research into the derivation of system specifications from requirements; but these specifications in the solution space are informal (such as UML sequence diagrams, Use Cases, Reusable Aspect Models, RSL-IL, and even natural language [9, Appendix 1, [10]). Other facets such as requirements elicitation and requirements validation methods are much less tackled, and traceability is seldom discussed. Many requirements models are based on variants of UML [11].

In this paper, we describe a method and tools for deriving specification models in the solution space from requirements, where the derived specification model (an executable abstract state machine1) is more formal than those described above, and thus amenable to analysis and validation. Validation involves running acceptance tests (derived from the requirements) on the model to check that user input-output relations and system safety properties (identified in the requirements) are satisfied.

Our tools (Fig. 1) go some way to making the process of keeping models and code in sync more seamless, and also to allowing for tracing requirements into the specification model.

- In section II, we introduce a small illustrative example called EHealth, to be used in the rest of the paper. EHealth is an electronic health system to ensure that patients’ medication prescriptions are safe. We document the requirements as numbered atomic descriptions.
- In section III, we describe the use of ETF (Eiffel Testing Framework) to specify an abstract 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

1 We use the term “abstract state machine” in the standard sense of a machine operating on states that are described by mathematical data structures. This is related to but not the same as the ASM method [12].
the design (the business logic). The use cases become acceptance tests when the final product is completed, and these tests check contracts in the business logic, as well as the correctness of sequences of feature calls.

- In section IV-A, we describe the use of the Mathmodels library for specifications. A specification of a system or a system component uses mathematical models (sets, sequences, relations, functions, bags) to describe an abstract state machine using contracts (preconditions, post-conditions, class invariants) in the Eiffel programming language. Classical contracts are incomplete or are low level implementation assertions. Mathmodels contracts provide complete specifications of components and systems, which scale up to validating large systems via runtime contract checking. Mathmodels have immutable queries (for specifications) as well as analogous mutable commands (for making the model executable and thus amenable to acceptance testing). Efficient code can be derived from an executable abstract state machine (the specification model) and kept in sync with it.

- In section IV-B, we complete the ETF generated code for the business logic with Mathmodels specifications derived from the requirements. The use cases (from the earlier phase) are used for acceptance testing of the software product. As the acceptance tests are run, the Mathmodels contracts are checked thus validating the correctness of the model. Class invariants encoded using Mathmodels ensures the safety of the system. Traceability is preserved between the numbered atomic requirements and the Mathmodels contracts. In this way, the program text retains important system consistency and safety properties, traced back to the original requirements.

Finally we compare our tools 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 large systems as contract checking is done automatically at runtime. We also discuss the relevance of this method and tools to computer science and software engineering education.

II. REQUIREMENTS ELICITATION

Specification is one of a trio of terms: requirements; specifications; and programs (Jackson, [13]). 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.

Writing a good requirements 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 requirements document if they cannot clearly identify what they have to take into account and in which order.

Important points may be missing or vague, and on the other hand, the requirements 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 running case study in this paper 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. It is kept small to fit the page limit, but without implying a limitation on the size or complexity of the systems which our tools and methods can handle.
We follow Jackson [13] and divide the requirements document into atomic ENV-descriptions (environmental constraints or assumptions) and REQ-descriptions (what the machine must produce). Elicitation of informal requirements produces the following:

<table>
<thead>
<tr>
<th>ENV1</th>
<th>Physicians prescribe medications to patients.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENV2</td>
<td>There exist pairs of medications that when taken together have dangerous interactions.</td>
</tr>
<tr>
<td>ENV3</td>
<td>If one medication interacts with another, then the reverse also applies (Symmetry).</td>
</tr>
<tr>
<td>ENV4</td>
<td>A medication does not interact with itself (Irreflexivity).</td>
</tr>
<tr>
<td>REQ5</td>
<td>The system shall maintain records of dangerous medication interactions.</td>
</tr>
<tr>
<td>REQ6</td>
<td>The system shall maintain records of patient prescriptions. No prescription may have a dangerous interaction.</td>
</tr>
<tr>
<td>REQ7</td>
<td>Physicians shall be allowed to add a medication to a patient’s prescription, provided it does not result in a dangerous interaction.</td>
</tr>
<tr>
<td>REQ8</td>
<td>It shall be possible to add a new medication interaction to the records, provided that it does not result in a dangerous interaction.</td>
</tr>
<tr>
<td>REQ9</td>
<td>Physicians shall always be allowed to remove a medication from a patient’s prescription.</td>
</tr>
</tbody>
</table>

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 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 [14].

Obviously, a real requirements document will contain many more numbered atomic descriptions organized hierarchically [8]. Labels (such as REQ8) are used later (Sec. IV-B) in the system models for traceability.

III. ETF: GENERATING AND TESTING MODELS VIA ABSTRACT USER INTERFACES

ETF (Eiffel Testing Framework) is an MDD tool for generating code from an abstract grammar specification of user interface actions derived from the requirements document (e.g. for EHealth described in Sec. II). The generated code is able to parse and execute acceptance tests (based on use cases derived from the requirements) that reference the UI actions.

Fig. 2 (p.4) is an example of a grammar specification for the EHealth system. Based on the requirements (Sec. II), we specify a grammar for the user input to the system. One may use a variety of basic types such as INT, VALUE (decimals with arbitrary precision), CHAR, and enumerations (e.g., KIND and PHYSICIAN). Composite types—tuples and arrays of tuples—may be recursively constructed from basic types. We may also declare synonyms to existing types. For example, in Fig. 2, a type MEDICATION is defined as:

\[
\text{TUPLE} \quad \{\text{name: NAME; kind: KIND; low: VALUE; hi: VALUE}\}
\]

The abstract grammar also defines possible user input actions, such as adding medications, physicians, interactions, etc. Comments are preceded by double dashes (--)..

For illustration, we will use the grammar specification for a smaller system in Fig. 3 (p.4). Based on the grammar, users may write acceptance tests even before the development of the business model (described in Sec. IV-B). Each acceptance test consists of a sequence of user actions such as adding medications and prescriptions.

Developers also need to describe the output after each action (with the help of their customers). Developers might want to think in terms of the abstract state which for the EHealth example is the prescriptions relation between patients and medications \( \subseteq \text{PATIENT} \times \text{MEDICATION} \), and the set of all dangerous interactions between medications given by \( \text{interactions} \subseteq \text{MEDICATION} \times \text{MEDICATION} \).

\(^2\) [MDD] Tools can automate the initial transformation [from model to code], and can help to keep the design and implementation models in step as they evolve. Typically the tools generate code stubs from the design models that the user has to further refine. As changes are made to the code they must at some point be reconciled with the original model” [15, p. 5].
Table I (p.10) is an example of an acceptance test for the EHealth system, where the abstract state is written in an ASCII format by the requirements engineer so that non-technical customers can understand the use case as well. For example, the set prescriptions: \{p1->m1,m3; p3->m2\} in state 16 means that patient p1 has been prescribed medications m1 and m3, and patient p3 has ben prescribed medication m2. In this acceptance test, we add medications, physicians and dangerous interactions. We also prescribe medications for patients.

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. As an example, consider the abstract state state 16 in Table I, where medication m2 interacts with m4 (i.e., the pair m2 \(\rightarrow\) m4 is a member of the interactions set). Thus, given that medication m2 is already prescribed for patient p3 in state 16, in state 17 a doctor cannot prescribe medication m4 for patient p3 because this would be dangerous for the patient. The use case then continues as follows: the interaction m2 \(\rightarrow\) m4, and symmetrically the interaction m4 \(\rightarrow\) m2, are removed at state 18, so that the prescription p3 \(\rightarrow\) m4 can be added in the subsequent state.

IV. USING MATHMODELS FOR SPECIFICATIONS

A. Overview of Mathmodels

In this section we provide a small illustrative example of the use of Mathmodels, and then explain why this modelling method scales up.

A software specification normally describes the set of services a system or component is expected to provide. A sorted map, for example, has features such as insert, remove and sorted_keys as shown in the class diagram of Fig. 5 (p.6). The specification must be precise so that it can act as a contract between the client and the supplier (understandable by both). A specification is also an abstraction, i.e. it should describe the important aspects and omit the unimportant ones.

Abstract State: In Fig. 5 and the corresponding Eiffel program text in Fig. 6 (p.6), sorted maps are specified using a mathematical model (a function from keys to values, \(\text{FUN}[\text{KEY}, \text{VALUE}]\)) to describe the abstract state. The features are specified in terms of the abstract state using preconditions, postconditions, and class safety invariants. For example, the first invariant (keys_are_sorted) specifies that keys of any sorted map can be accessed as a sorted sequence.
Executable Abstract State Machines: Mathmodels classes such as FUN have immutable queries for contracts and analogous mutable commands for making the specifications executable (which may be refined to more efficient descendants). In Line 28 of Fig. 6 (p.6), insert is implemented via the override_by command, which is the mutable analogue of function overriding (with infix symbol “+”). Executability of the model means that the model can be validated—before developing efficient descendants that are required to conform to the model. We may thus consider class SORTED_MAP with its model-based contracts as an abstract state machine.

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 sorted map, we use the same notation to express specifications and implementations within the same syntactic and semantic universe (Eiffel in this case). In an ideal world where requirements are fixed at the start, one might switch notations between models and code. 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.

Systems specified by Mathmodels are developed and validated using the industrial strength Eiffel IDE [16], and the use of these models thus scale up to the development of large systems in a way that supports seamlessness between models and code.

A method to achieve demonstrable correctness is via mathematical proofs performed mechanically, but for large systems this is still work in progress requiring advanced expertise. Runtime assertion testing (rather than proof-based methods) has been perfected on the Eiffel IDE over several decades and used daily for large-scale mission-critical applications. The approach is incremental. Unlike fully formal methods and proofs, it does not require one to write down every single property down to the last quantifier. One may start with simple contracts. The more we write, the more we get; it is the opposite of an all-or-nothing approach.

On the practical side, there are no compromises on the performance of a delivered system. Runtime contract monitoring is a compilation option, tunable for different kinds of contracts (invariants, and pre/postconditions) and different parts of a system. The contracts are used for development, testing and debugging, and may be turned off on production systems.4

B. EHealth Specification

As explained earlier (Sec. IV-A), the Mathmodels library has immutable queries for models specified by tuple, sets, sequences, functions, relations and bags. It also has analogous mutable commands for making models executable.

---

3For documentation of Mathmodels, see http://www.eecs.yorku.ca/course_archive/2016-17W/3311/eiffel-docs/mathmodels/index.html. The Library is available as open source at https://svn.eecs.yorku.ca/repos/sel-open/mathmodels.

In Sec. III, we described how ETF generates Eiffel code with a placeholder for developing a model of the business logic for a software product.

In this section, we illustrate the use of the Mathmodels library to provide executable models (as Eiffel program text) for the business logic of the EHealth example (Sec. II).

We develop the business logic in the Model package of the generated code as shown in Fig. 4 (p.5), where we add model classes such as MEDICATION and INTERACTION (see Fig. 7). Class HEALTH_SYSTEM in Fig. 9 (p.7) describes the abstract state of the EHealth application (Lines 4 to 11), now formalized as Eiffel text amenable to compiler syntax, type and runtime assertion checking, e.g.:

- prescriptions with type REL[ PATIENT, MEDICATION ] is a relation between patients and medications. Class REL[ G, H ] as shown in Fig. 8 is part of the Mathmodels library.
- interactions with type SET[ INTERACTION ] is a set of interactions.

Importance of system safety invariants: From requirements elicitation (Sec. II), we identified important system constraints such as REQ6 asserting that patients are never prescribed dangerous interactions. These requirements become part of the model as invariants in class HEALTH_SYSTEM (Fig. 9, p.7):
8th International Workshop Model-Driven Requirements Engineering (MoDRE), August, 2018

### Requirement ENV3 (symmetry) shown at Lines 13–18;

- Requirement ENV4 (irreflexivity) shown at Lines 19–22;
- Requirement REQ6 shown at Lines 23–31.

In standard mathematical notation, for medications $m_1$ and $m_2$

$$
\forall p \in pr.\text{domain}, m_1, m_2 : \\
\frac{m_1 \neq m_2 \land (m_1, m_2) \in \text{interactions}}{\exists \{p, m_1\} \in pr \land \exists \{p, m_2\} \in pr}
$$

where $pr$ stands for the prescriptions relation. The Eiffel text is more verbose but encodes the same specification.

These invariants ensure that any actions performed by users will preserve the integrity of the data and the safety of patients. In order to preserve each of these crucial invariants, the actions (user inputs) must have preconditions that are guaranteed to ensure the invariants.

**Traceability:** By documenting the informal requirements as numbered atomic descriptions (e.g. REQ6) we can trace where the model formalizes the requirements.

In the EHealth system, each user interface action (e.g. add_prescription and add_interaction) has a corresponding model class (ADD_PRESCRIPTION in Fig. 10 and ADD_INTERACTION in Fig. 11) that inherits the abstract state and safety invariants from HEALTH_SYSTEM.

**Invariants drive the derivation of preconditions:** Consider the precondition of command add_prescription$(p, m)$ in class ADD_PRESCRIPTION starting at Line 6 of Fig. 10:

- Line 10 asserts that patient $p$ must be in the system and Line 12 asserts that medication $m$ is not yet prescribed for patient $p$. The query prescriptions$[p]$ (from class REL in Fig. 8, p. 6) is the relational image returning a set of medications for $p$.
- Lines 15 to 17 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.

Classes such as ADD_PRESCRIPTION in the Model package are given demanding preconditions, whereas the
The ETF Tool described in this paper does not seem to have an analogue in the literature. It is used at the transition from requirements in the problem domain to a specification of the abstract user interface in the solution domain, the derivation of acceptance tests, and model-ready code generation. Of course, modern IDEs contain a sophisticated “design” perspective where one graphically specifies a concrete user interface, and a “text” perspective which is generated automatically from the design (e.g. in XML). But these interfaces have to provide additional implementation details—beyond the functional actions that the UI must support; details such as the placement and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layouts. Also, in writing acceptance tests for regression testing, the test scripts are written in a programming language referring to details such as which browser to to open, which identifiers to access and organization of the widgets, representation issues (drop down menu vs. radio buttons) and the various layout...
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 tens of thousands of lines of code. An advantage is that if the verification succeeds, then we have a proof of correctness, that transcends what testing can do. However, manual intervention is often required and expertise is needed. By contrast, in runtime checking such as in Eiffel, proofs are lacking but large systems can be handled for verification. Manual intervention is not needed as it is in theorem proving.

**Complementing Testing with more Formal Methods**

We have manually transformed models developed with Mathmodels into TLA+ specifications [27], [28]. This is relatively simple to do as there are analogous TLA+ constructs for Mathmodels sets, functions, relations, etc. To give a simple example, the REQ6 class invariant in Fig. 9 translates to the predicate shown in equation (1, p.7), which is close to the TLA+ encoding. The TLA+ modelchecker (called TLC) can then check the model automatically as described in [17]. Other formal methods may also be used. It is possible to automate the transformation from Mathmodels to TLA+, or languages of other theorem proving or modelchecking tools.

**Computer Science and Software Engineering Education**

As pointed out in [2], the challenges of MDE adoption has a “pedagogic/training nature”. Industry representatives have reported the difficulty of hiring well-trained MDE practitioners.

We 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 acceptance tests provided as part of the specification.

In an article titled “Teach Foundational Language Principles: Industry is ready and waiting for more graduates educated in the principles of programming languages”, the authors make some recommendations for computer science education looking to the future [29]. The authors are Thomas Ball, 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.

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 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. Ball and Zorn write [29]:

“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.”

**References**

<table>
<thead>
<tr>
<th>Table I</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ETF Use Case: add dangerous interactions and prescriptions</strong></td>
</tr>
</tbody>
</table>

```
state 0
patients: {}
interactions: {}
prescriptions: {}

→add_patient("p1")
state 1
patients: {p1}
medications: {}
interactions: {}
 prescriptions: {}

... 

→add_patient("p3")
state 3
patients: {p1, p2, p3}
medications: {m1, m2, m3, m4}
interactions: {m1->m2, m2->m1, m2->m4, m4->m2}
prescriptions: {}

... 

→add_interaction("m1", "m2")
state 10
patients: {p1, p2, p3}
medications: {m1, m2, m3, m4}
interactions: {m1->m2, m2->m1, m2->m4, m4->m2}
prescriptions: {}

... 

→add_medication("m1")
state 5
patients: {p1, p2, p3}
medications: {m1}
interactions: {}
prescriptions: {}

... 

→add_interactions("m1", "m2")
state 11
patients: {p1, p2, p3}
medications: {m1, m2, m3, m4}
interactions: {m1->m2, m2->m1, m2->m4, m4->m2}
prescriptions: {}

... 

→add_management("m1")
state 12
patients: {p1, p2, p3}
medications: {m1}
interactions: {m1->m2, m2->m1, m2->m4, m4->m2}
prescriptions: {}

... 
```