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}@eecs.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].

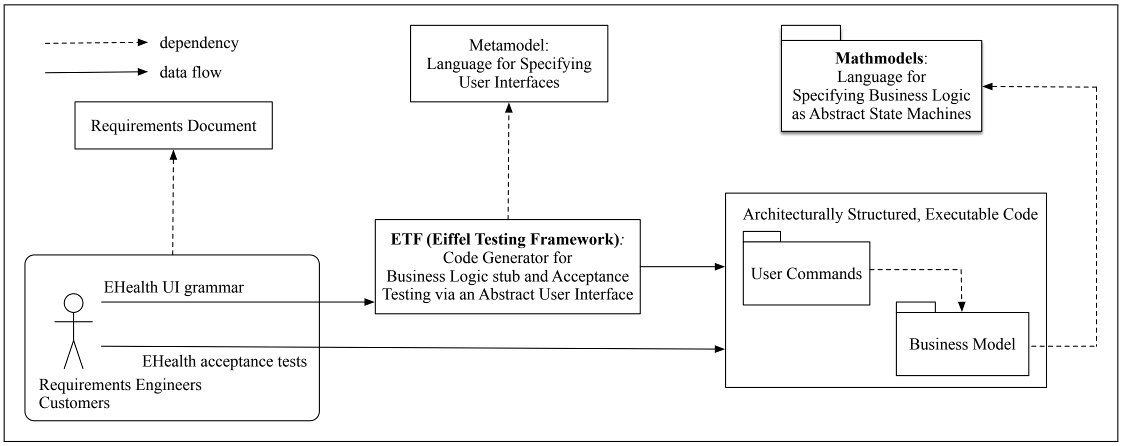
In this paper, we describe a method and tools for deriving specification models in the solution space from requirements, where the derived specification model (an executable abstract state machine[1]) is more formal than those described above, and thus amenable to analysis and validation. Validation 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].

Fig. 1. ETF MDD Tool and *Mathmodels* Library



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, postconditions, 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:

ENV1	Physicians prescribe medications to *patients*.

ENV2	There exist 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 (Irreflexivity).

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 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 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 REQ6) 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).[2] 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:

TUPLE [*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 $prescriptions \subseteq PATIENT \times MEDICATION$, and the set of all dangerous interactions between medications given by $interactions \subseteq MEDICATION \times MEDICATION$.

[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].

Fig. 2. ETF Abstract Grammar for an EHealth System

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*)
...

Fig. 3. Abstract Grammar for a Smaller EHealth System

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*))

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 the various 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 -> 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 -> m4`, and symmetrically the interaction `m4 -> m2`, are removed at `state 18`, so that the prescription `p3 -> m4` can be added in the subsequent state.

Use of the ETF Tool is as follows:

- Based on the requirements (such as Sec. II), specify the grammar for an abstract user interface (Fig. 3) of data types and user inputs to the system. There is no need to commit, prematurely, to a concrete user interface.
- Once the grammar is specified, the developers may write acceptance tests to validate the software.
- The ETF tool is invoked on the grammar to generate architecturally structured code (see Fig. 4, p.5) using the command design pattern.
  - Develop the business logic (described later in Sec. IV-B) in the Model package.
  - The User Commands package contains classes (e.g., *ETF_ADD_PATIENT*) associated with the user interface. As shown in the UML diagram in Fig. 4, 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).
  - Integrate the business model with the user interface. For example, the descendant class *ETF_ADD_PATIENT* from the User Commands package either signals an error if the input patient already exists, or invokes the relevant model action.
  - Acceptance tests (e.g., Table I, p.10) may be executed from the command line. The generated code will report syntax and type errors (if any) or display the expected output if the business logic is correct.
- Once the business logic is developed using the *Mathmodels* library (Sec. IV-B), running the acceptance tests (e.g., Table I) also verifies that relevant classes satisfy their specifications (preconditions, postconditions, class invariants).

IV. Using Mathmodels for specifications

A. Overview of Mathmodels

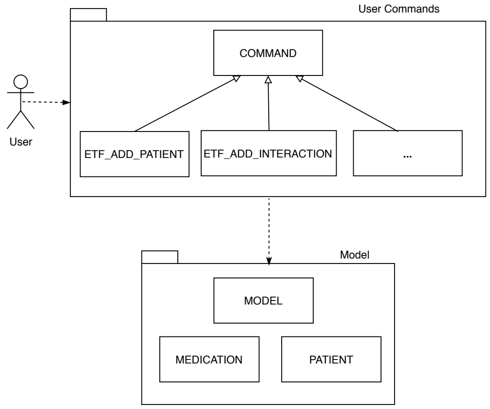
In this section we provide a small illustrative example of the use of Mathmodels, and then explain why this modelling method scales up.

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, *FUN*[*KEY*, *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.

Fig. 4. Generated Code by the ETF Tool

Class Diagram: User Commands package depends on Model package



Consider the generated Eiffel file `etf_add_patient.e` for example:

```
class ETF_ADD_PATIENT inherit COMMAND feature
  add_patient(p: STRING)
    local l_p: PATIENT
    do
      create l_p.make (p) -- create a patient from model package
      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 ...
    end
```

The Model package is a placeholder where the software developer will describe the business logic, by introducing model classes such as PATIENT and MEDICATION etc., and using Mathmodels to specify abstract state variables and state changes to user inputs.

By contrast, sorted maps can be implemented in many different ways, e.g., using sorted arrays or red-black trees. These code implementations will introduce additional lower level constructs such as nodes, pointers and trees. However, these implementations inherit all the model specifications of class *SORTED_MAP* and must thus satisfy its abstract preconditions, post-conditions and safety invariants. Implementations may change, but the specifications remain the same. This is the power of abstraction.

Mathmodels[3]: The class *FUN*[*KEY*,*VALUE*] representing mathematical functions is part of our Mathmodels library which also contains tuples, sets, relations, bags, etc (encoded in Eiffel). Thus, for example, the *insert(key, val)* command in Fig. 6 is specified with a precondition asserting that $key \notin model.domain$ and a postcondition $model =$ **old** $model \cup (key, val)$. The Eiffel notation for these contracts is shown in Lines 27 and 29 of Fig. 6 (p.6). These contracts specify how the abstract state changes when an insert operation is performed.

[3]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.

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.

[4]See [17] for more Mathmodels examples, and https://bertrandmeyer.com/2018/05/24/not-program-right/, accessed 2018-05-24

Fig. 5. Safety Invariants of Sorted Maps

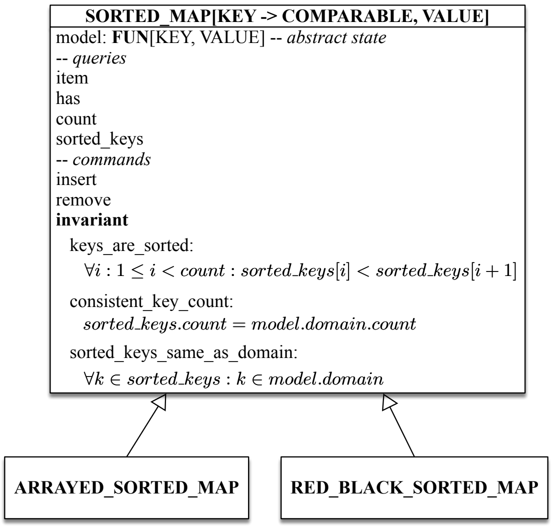


Fig. 6. An Eiffel Abstract State Machine for a Sorted Map

```
class SORTED_MAP [K −> COMPARABLE, V] feature
  model: FUN [K, V] -- abstract state
    do Result := model_imp end
  model_imp: like model -- to make the model executable
feature -- queries
  item alias ‘‘[]’’ (key: K): V
      -- get value for ‘key’
    require has (key)
    do Result := model_imp [key]
    ensure Result = model [key] end
  sorted_keys: ARRAY [K]
      -- return a sorted array of keys
    do Result := model_imp.domain.as_array
       Result := {SORT [K]}.quicksort (Result)
    ensure -- see the invariant end
  has (key: K): BOOLEAN
      -- does key/value pair exist?
    do Result := model_imp.domain.has (key)
    ensure Result = model.domain.has (key) end
  count: INTEGER
      -- number of elements in the map
    do Result := model_imp.count
    ensure Result = model.count end
feature -- commands
  insert (key: K; val: V)
      -- insert ‘key’ and ‘val’
    require key_unique: not has (key)
    do model_imp.override_by ([key, val])
    ensure model ∼ ((old model) + [key, val]) end
  remove (k: K)
      -- remove key ‘k’ and associated value
    require has (k)
    do model_imp.subtract ([k, model_imp[k]])
    ensure model ∼ ((old model) − [k, old model[k]]) end
invariant
  keys_are_sorted:
    across 1 |..| (model.count − 1) as i
      all sorted_keys [i.item] < sorted_keys [i.item + 1] end
  consistent_key_count:
    sorted_keys.count = model.domain.count
  sorted_keys_same_as_domain:
    across sorted_keys as k
      all model.domain.has (k.item) end
end
```

Fig. 7. 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
```

Fig. 8. Some Queries of Mathmodels Class 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
    ...
end
```

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 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):

Fig. 9. class HEALTH_SYSTEM with Abstract State and Invariants

```
1  class
2    HEALTH_SYSTEM
3  feature -- abstract state
4    patients: SET [PATIENT]
5      -- set of patients
6    medications: SET [MEDICATION]
7      -- set of medications
8    prescriptions: REL [PATIENT, MEDICATION]
9      -- prescriptions
10   interactions: SET [INTERACTION]
11     -- dangerous interactions
12 invariant
13   symmetry_ENV3:
14     across medications as m1 all
15     across medications as m2 all
16       interactions.has (m1.item ↦ m2.item)
17       = interactions.has (m2.item ↦ m1.item)
18     end end
19   irreflexivity_ENV4:
20     across medications as m1 all
21       not interactions.has (m1.item ↦ m1.item)
22       end
23   no_dangerous_interactions_REQ6:
24     across prescriptions.domain as p all
25     across prescriptions[p.item] as m1 all
26     across prescriptions[p.item] as m2 all
27       interactions.has (m1.item ↦ m2.item)
28       implies
29       not(prescriptions.has([p.item,m1.item])
30           and prescriptions.has([p.item,m2.item]))
31     end end end
32   consistent_domain:
33     prescriptions.domain ⊆ patients
34 end
```

Fig. 10. class ADD_PRESCRIPTION with Pre- and Post-Conditions

```
1  class
2    ADD_PRESCRIPTION
3  inherit
4    HEALTH_SYSTEM
5  feature -- commands
6    add_prescription (p: PATIENT; m: MEDICATION)
7            -- Add a prescription of 'm1' to 'p1'.
8      require
9        -- p ∈ patients
10       patients.has (p)
11       -- m ∉ prescriptions[p]
12       not prescriptions[p].has (m)
13       -- cannot cause a dangerous interaction
14       -- ∀med ∈ prescriptions[p] : (med, m) ∉ interaction
15       across prescriptions[p] as med all
16         not interactions.has(med.item ↦ m)
17       end
18     do
19       prescriptions.extend ([p, m])
20     ensure
21       prescriptions ∼ old prescriptions + [p, m]
22       -- UNCHANGED (patients, medications, interactions)
23   end
24 end
```

Fig. 11. class ADD_INTERACTION with Pre- and Post-Conditions

```
1  class
2    ADD_INTERACTION
3  inherit
4    HEALTH_SYSTEM
5  feature -- commands
6    add_interaction (m1, m2: MEDICATION)
7        -- Add an interaction between 'm1' and 'm2'.
8      require
9        medications.has (m1) and medications.has (m2)
10       m1 ≠ m2
11       not interactions.has (m1 ↦ m2)
12       -- ∀p ∈ dom(prescriptions) : {m1, m2} ⊈ prescriptions[p]
13       across prescriptions.domain as pc all
14         not (⟪ m1, m2⟫ ⊆ prescriptions[pc.item])
15       end
16     do
17       interactions.extend ([m1, m2])
18       interactions.extend ([m2, m1])
19     ensure
20         interactions ∼ old interactions + [m1, m2] + [m2, m1]
21         -- UNCHANGED (patients, medications, prescriptions)
22     end
23 end
```

- 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 and patient p, REQ6 is:

$$\begin{aligned} &\forall p \in pr.domain, m_1, m_2 : \\ &\quad m_1 \neq m_2 \wedge (m_1, m_2) \in interactions \\ &\quad \Rightarrow \neg((p, m_1) \in pr \wedge (p, m_2) \in pr) \end{aligned} \tag{1}$$

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 analogous classes in the User Commands package (e.g. *ETF_ADD_PRESCRIPTION*) invoked at the user interface (see Fig. 4) 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 valid 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 package). If it is not legal, it must signal to the external users that the inputs are problematic.

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 handling at the interface to the external world is to guarantee that no information is passed to the business logic that would cause inconsistent data.

Acceptance Testing: Design by Contract views a software system as a set of components whose collaboration is based on precisely defined specifications of mutual obligations—the contracts. As mentioned earlier (Sec. III, p.3), acceptance tests (as in Table I, p.10) are used to validate the integrated system. Also, when these acceptance tests are executed, they also exercise and thus verify the model contracts of the business logic.

V. Discussion

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 in the html (in a web application, for example), before actually getting to tests of the business logic.

The report [18] 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, we also used model libraries in Eiffel together with a theorem prover for proving properties [19] .

The authors of [20] 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 large systems because the checking is completely automatic, albeit without the completeness of theorem proving. Also, Mathmodels is Void safe [21] whereas the others are not.

Mathmodels may be used to specify concurrent systems using Eiffel's SCOOP mechanism [22], [23], but is not suitable for designing systems with hard real-time constraints.

There is a significant body of work on models, contracting mechanisms and their analysis. The paper [24] provides a survey of contracting mechanisms, comparing Eiffel with other frameworks developed for languages such as Java and C#. A major challenge is the use of theorem proving to scale up to large systems.

For example, [25] 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 [26]. It 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 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

[1] S. Kelly and J.-P. Tolvanen, *Domain-Specific Modeling*. John Wiley, 2007.

[2] M. Goulão, V. Amaral, and M. Mernik, "Quality in model-driven engineering: a tertiary study," *Software Quality Journal*, vol. 24, no. 3, pp. 601–633, Sep 2016.

[3] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice (Second Edition)*. Morgan & Claypool Publishers, 2017.

[4] J. Hutchinson, J. Whittle, and M. Rouncefield, "Model-driven engineering practices in industry: Social, organizational and managerial factors that lead to success or failure," *Science of Computer Programming*, vol. 89, pp. 144 – 161, 2014.

[5] J. Whittle, J. Hutchinson, M. Rouncefield, H. Burden, and R. Heldal, "A taxonomy of tool-related issues affecting the adoption of model-driven engineering," *Software and Systems Modeling*, vol. 16, no. 2, pp. 313–331, 5 2017.

[6] 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.

[7] F. L. Siqueira, P. S. M. Silva, and P. S. M. Silva, “Transforming an enterprise model into a use case model using existing heuristics,” in *2011 Model-Driven Requirements Engineering Workshop*, Aug 2011, pp. 21–30.

[8] E. Hull, K. Jackson, and J. Dick, *Requirements Engineering*. Springer Verlag, 2005.

[9] S. Assar, “Model driven requirements engineering: Mapping the field and beyond,” in *2014 IEEE 4th International Model-Driven Requirements Engineering Workshop (MoDRE)*, Aug 2014, pp. 1–6.

[10] G. Loniewski, E. Insfran, and S. Abrahão, “A systematic review of the use of requirements engineering techniques in model-driven development,” in *Model Driven Engineering Languages and Systems*, D. C. Petriu, N. Rouquette, and Ø. Haugen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 213–227.

[11] A. van Lamsweerde, *Requirements Engineering - From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[12] E. Borger and R. F. Stark, *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

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

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

[15] S. Beydeda, M. Book, and V. Gruhn, Eds., *Model-Driven Software Development*. Springer, 2005.

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

[17] J. S. Ostroff, *Validating Software via Abstract State Specifications*. EECS, Lassonde School of Engineering, York University, no. EECS-2017-02.

[18] 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.

[19] 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.

[20] 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.

[21] 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

[22] 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

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

[24] 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

[25] R. Leino and V. Wüstholz, “The Dafny Integrated Development Environment,” in *Workshop on Formal Integrated Development Environment (F-IDE)*, April 2014.

[26] 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

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

[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] 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

TABLE I
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
->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
->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","m2")
 state 16
 patients:      {p1,p2,p3}
 medications:   {m1,m2,m3,m4}
 interactions:  {m1->m2,m2->m1,m2->m4,m4->m2}
 prescriptions: {p1->m1,m3; p3->m2}
->add_prescription("p3","m4")
 state 17 Error e4: this prescription dangerous
->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}
```