Modelling and Testing Requirements via Executable Abstract State Machines

Model-Driven Requirements Engineering (MoDRE)
August 20, 2018 / Banff, Canada

Jonathan S. Ostroff and **Chen-Wei Wang**

Case Study: An E-Health System

- *Patients* are prescribed to *medications*.
- Medications may have *dangerous interactions*.

  e.g., warfarin and aspirin both increase anti-coagulation
- Goal: *No dangerous interactions* in patients' prescriptions.

image source: `https://www.canstockphoto.ca`

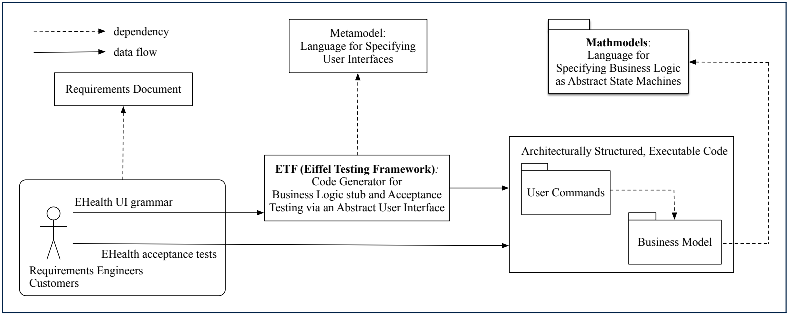
Research Problem

Given *informal* requirements describing the *problem domain*, how can we facilitate **the process** of developing *working* code in the *solution domain*?

We present a method for facilitating this process: from requirements to *formal*, *executable* specifications.

Contributions

- ***ETF*** (Eiffel Testing Framework)
  - Generates *code stub* for developing business logic
  - Supports *acceptance testing* via a given Abstract User Interface
- ***Mathmodels*** programming library
  - Specifies business logic as *abstract state machines*


dependency
data flow
Requirements Document
Metamodel: Language for Specifying User Interfaces
Mathmodels: Language for Specifying Business Logic as Abstract State Machines
Architecturally Structured, Executable Code
ETF (Eiffel Testing Framework): Code Generator for Business Logic stub and Acceptance Testing via an Abstract User Interface
User Commands
Business Model
EHealth UI grammar
EHealth acceptance tests
Requirements Engineers
Customers

- *Scalable to large systems* via *Runtime Contract Checking*.

Requirements Elicitation (1)

ENV-descriptions document environment constraints or assumptions.

ENV1	Physicians prescribe medications to *patients*.
ENV2	There exist pairs of medications that when taken together have dangerous *interactions*.
ENV3	If one *medication* interacts with another, then the reverse also applies (Symmetry).
ENV4	A medication does not interact with itself (Irreflexivity).

reflected in Mathmodels

Requirements Elicitation (2)

REQ-descriptions document what the machines must produce.

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

reflected in Mathmodels

Abstract User Interface

```
system ehealth
  -- semantics types
  type MEDICATION = STRING
  type PATIENT = STRING
  -- events
  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))
```

Abstract UI may **later** be implemented using concrete desktop, mobile, or web interface.

Abstract State

Types of ***abstract*** state variables:

$$\begin{array}{lcl} \textit{patients} & \in & \mathbb{P}\,\textit{PATIENT} \\ \textit{medications} & \in & \mathbb{P}\,\textit{MEDICATION} \\ \textit{interactions} & \in & \textit{MEDICATION} \leftrightarrow \textit{MEDICATION} \\ \textit{prescriptions} & \in & \textit{PATIENT} \leftrightarrow \textit{MEDICATION} \end{array}$$

Example ***abstract state*** in ASCII form:

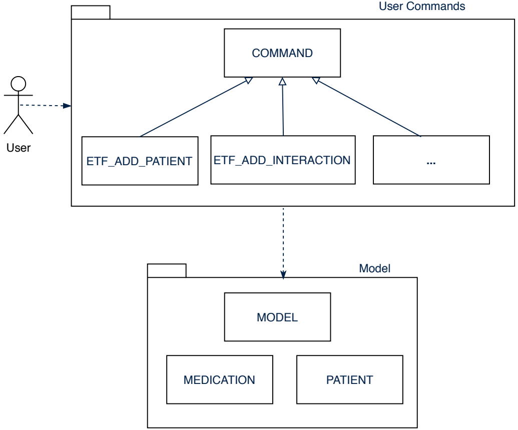
```
patients:      {p1, p2, p3}
medications:   {m1, m2, m3, m4}
interactions:  {m1 -> m2, m2 -> m1}
prescriptions: {p1 -> m1, m3; p3 -> m2,m4}
```

Acceptance Test

```
...
 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}
```

Architecturally Structured Generated Code

- Given an *abstract UI*, ETF generates *architecturally structured code*.



- *Business logic* is specified and implemented in the `MODEL` package.
- *Error handling* is implemented in the `User Commands` package.

The Mathmodels Library

```
class
  REL[ G, H ]
inherit
  SET[ TUPLE[ G, H ] ]
feature -- immutable queries
  domain: SET[ G ]
  range: SET[ H ]
  image alias "[]" (g: G): SET[ H ]
  extended alias "+" (p: TUPLE[ G, H ]): REL[ G, H ]
  overriden_by (p: TUPLE[ G, H ]): REL[ G, H ]
feature -- mutable commands
  extend (p: TUPLE[ G, H ])
  override (p: TUPLE[ G, H ])
     ...
end
```

- *Immutable queries* for specifying *precise contracts*.
- *Mutable commands* for making *executable Abstract State Machine*.
- There are other classes in Mathmodels library: `SET`, `FUN`, `BAG`.

Mathmodels vs. Math

- Recall the *informal* R-description:

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

- How to *formulate* it using set theory and predicate logic?

$$\begin{aligned}&\forall p \in \mathit{patients};\ m_1, m_2 \in \mathit{medications} :\\ &\quad p \in \mathrm{dom}(\mathit{prescriptions}) \land m_1 \neq m_2 \land (m_1, m_2) \in \mathit{interactions}\\ &\qquad \Rightarrow \neg(\ (p, m_1) \in \mathit{prescriptions} \land (p, m_2) \in \mathit{prescriptions}\ )\end{aligned}$$

- How to make the above formula *executable* and *traceable*?

```
no_dangerous_interactions_REQ6 :
  across prescriptions.domain as p all
  across prescriptions[p.item] as m1 all
  across prescriptions[p.item] as m2 all
    interactions.has ( [m1.item, m2.item] )
    implies
    not( prescriptions.has( [p.item, m1.item] ) and prescriptions.has( [p.item, m2.item] ) )
  end end end
```

Using Mathmodels to Contract Abstract State

Invariants are *traceable* back to ENV- and REQ-descriptions.

```
class
  HEALTH_SYSTEM
feature -- abstract state
  patients: SET [PATIENT]
  medications: SET [MEDICATION]
  prescriptions: REL [PATIENT, MEDICATION]
  interactions: SET [INTERACTION]
invariant
  symmetry_ENV3:
    across medications as m1 all
    across medications as m2 all
      interactions.has ( [m1.item, m2.item] ) = interactions.has ( [m2.item, m1.item] )
    end end
  irreflexivity_ENV4:
    across medications as m1 all not interactions.has ( [m1.item, m1.item] ) end
  no_dangerous_interactions_REQ6:
    across prescriptions.domain as p all
    across prescriptions [p.item] as m1 all
    across prescriptions [p.item] as m2 all
      interactions.has ( [m1.item, m2.item] )
        implies not( prescriptions.has( [p.item,m1.item] ) and prescriptions.has( [p.item,m2.item] ) )
    end end end
  consistent_domain:
    prescriptions.domain ⊆ patients
end
```

Using Mathmodels to Contract Actions

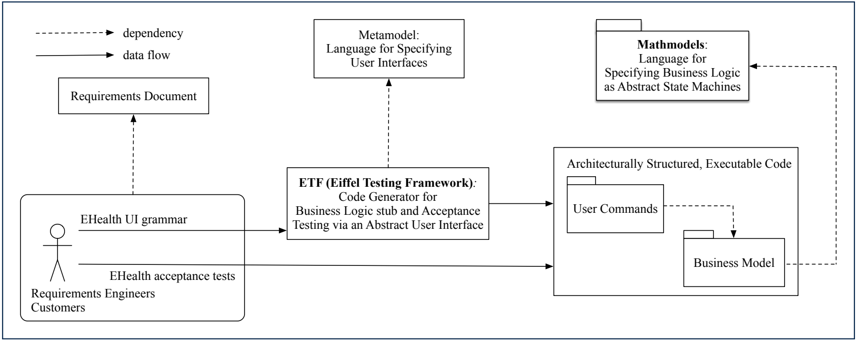
State updates are contracted with *pre-conditions* and *post-conditions*.

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

```
class
  ADD_PRESCRIPTION
inherit
  HEALTH_SYSTEM -- inherits all system invariants
feature -- commands
  add_prescription (p: PATIENT; m: MEDICATION)
        -- Add a prescription of 'm1' to 'p1'.
    require
      -- p ∈ patients
      patients.has (p)
      -- m ∉ prescriptions[p]
      not prescriptions[p].has (m)
      -- cannot cause a dangerous interaction
      -- ∀med ∈ prescriptions[p] : (med, m) ∉ interaction
      across prescriptions[p] as med all not interactions.has( [med.item, m] ) end
    do
      prescriptions.extend ([p, m])
    ensure
      prescriptions ~ old prescriptions + [p, m]
      -- UNCHANGED (patients, medications, interactions)
    end
end
```

Summary

- ***ETF*** (Eiffel Testing Framework) [code generator]
- ***Mathmodels*** programming library [specification language]



dependency
data flow
Requirements Document
Metamodel: Language for Specifying User Interfaces
Mathmodels: Language for Specifying Business Logic as Abstract State Machines
ETF (Eiffel Testing Framework): Code Generator for Business Logic stub and Acceptance Testing via an Abstract User Interface
Architecturally Structured, Executable Code
User Commands
Business Model
EHealth UI grammar
EHealth acceptance tests
Requirements Engineers
Customers

The proposed method adopts *Design-by-Contract* (DbC) and *Eiffel programming IDE*.

⇒ *Scalable to large systems* via *Runtime Contract Checking*.

Index (1)