Combining Formal Methods and MDE Techniques for Model-driven System Design and Analysis

Elvinia Riccobene
elvinia.riccobene@unimi.it

Dipartimento di Tecnologie dell’Informazione
Università degli Studi di Milano, Italy
PIRE project

• I am member of the steering committee of the PhD Program at the University of Milan

• PIRE agreement foresees:
  - Exchange of PhD Students between FAU and UNIMI
  - For a total of 6 months per year in each direction
FM&SE_lab@unimi: Formal methods and Software Engineering Lab

• Composition:
  - 2 faculty members
  - 1 Ph.D student
  - External people:
    • Strong collaboration with the University of Bergamo

• Research topics:
  Formal methods and their practical applications in software engineering
  • **Specification:**
    - Formal specification using Abstract State Machines
    - Integration between formal and model-driven system specification methods
    - Formal modeling of Service-Oriented Applications
  • **Analysis:**
    - Formal validation and scenario-cased validation
    - Model review by model advisor
    - NuSMV Model checking
    - Model-based testing
    - Run-time monitoring of Java code
      » combination of the last two
Topic of the talk


Other references:

Outline

- Advantages and disadvantages of FMs and MDE for SW system development
- Why and how to integrate FMs and MDE
- Description of the activities
  - MDE for FMs
  - FMs for MDE
- Instantiation with EMF and ASMs in
  - in-the-loop integration
- Conclusions
Formal Methods in SE

• Formal methods are intended to systematize and introduce rigor into all the phases of software development
• FMIs are a particular kind of mathematically-based techniques for the specification, development (by refinement) and verification of software and hardware systems
• As in other engineering disciplines, performing appropriate mathematical analysis can contribute to the reliability and robustness of a design
  - especially for high-integrity systems where safety or security need to be formally proved
Model Driven Engineering

• A new software development methodology based on the concepts of:
  - models as first class artifacts
  - metamodeling to define languages for modeling
  - automatic model transformations
    • to transform models into other models or reduce models into code

• Well established process in the context of domain-specific language development

• Many MDE frameworks and supporting technologies
  - EMF/Ecore, OMG/MOF, AMMA/KM3,...
Two contexts in SW system development

<table>
<thead>
<tr>
<th>ADVANTAGES</th>
<th>DISADVANTAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Based on rigorous mathematical foundation</td>
<td>• Complex notation</td>
</tr>
<tr>
<td>• Suitable for model analysis (validation &amp; verification)</td>
<td>• Lack of tools</td>
</tr>
<tr>
<td>• User-friendly notation</td>
<td>• Lack of integration</td>
</tr>
<tr>
<td>• Derivative artifacts for tool development</td>
<td></td>
</tr>
<tr>
<td>• Model transformation (autom.)</td>
<td></td>
</tr>
<tr>
<td>• code generation from models</td>
<td>• Lack of semantics</td>
</tr>
<tr>
<td>• Inadequate for model analyses</td>
<td></td>
</tr>
</tbody>
</table>

**Formal Methods**

- Models + refinement/transform.

**Model-driven Engineering**
## Integration Envisioning

<table>
<thead>
<tr>
<th>ADVANTAGES</th>
<th>DISADVANTAGES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Formal Methods</strong></td>
<td><strong>Model-driven Engineering</strong></td>
</tr>
<tr>
<td>• Based on rigorous mathematical foundation</td>
<td>• Complex notation</td>
</tr>
<tr>
<td>• Suitable for model analysis (validation &amp; verification)</td>
<td>• Lack of tools</td>
</tr>
<tr>
<td>• User-friendly notation</td>
<td>• Lack of integration</td>
</tr>
<tr>
<td>• Derivative artifacts for tool development</td>
<td>• Inadequate for model analyses</td>
</tr>
<tr>
<td>• Model transformation</td>
<td>• Inadequate for model generation from models</td>
</tr>
</tbody>
</table>
**MDE for FMs**

**IMPROVEMENTS**
- Intuitive *modeling notation*
  - possible graphical view of models
- Open and flexible *architecture*
  - for tools dev. & integration

**DISADVANTAGES**
- Complex notation
- Lack of tools
- Lack of integration

**Formal Methods**

**Model-driven Engineering**
- User-friendly notation
- Derivative artifacts for development
- Model transformation
- Automatic code generation from models
FM for MDE

Improvements

- Based on rigorous mathematical foundation
- Suitable for model analysis (validation & verification)
- Rigorous semantics of MDE models
  - Techniques & methods for formal analysis of models
    - Simulation, testing, proofs, model checking, etc.

Disadvantages

- Lack of semantics
  - Inadequate for model analyses
Activity 1: MDE for FM

- It has the overall goal of engineering a language and a tool-set around the formal method in order to support its practical use in systems development life cycle.

- In practice, *this activity* consists mainly of:
  - designing the abstract syntax of the formal language by building a **metamodel**
  - defining language **concrete syntaxes**
  - developing **tools** to process and analyze models
Activity 2: FM for MDE

• It has the overall goals of
  1. allowing the definition of the behaviors (semantics) of models conforming to the MDE language L and
  2. providing techniques for formal analysis of models conforming to L
     • Validation (simulation, scenario generation, model-based testing, etc.)
     • Verification (formal proofs, model checking)

• In practice, this activity consists of:
  - selecting an helper formal language
  - defining the semantics to the metamodel-based language L by following this schema
Activity 2: FM for MDE

- Let $L$ be an MDE language with a metamodel $A$
  - to formally define semantics of $L$, we have to fix:
    - a semantic domain $S$ and
    - a semantic mapping $M_S: A \rightarrow S$
- Suppose to have a $L' = <A', M'_{S'}, >$ (helper language)
  - where $S'$ and $M'_{S'}$ well-defined
- Assume $S' = S$

$M$: building function

Semantic model of $m$
Characteristics of an Helper Language

• be **formal** to rigorously define model behavior at different levels of **abstraction**, but without formal overkill

• be able to capture different models of computation (**MoC**)

• be endowed with a model **refinement** mechanism

• be endowed with a **metamodel** to exploit the automatic models mapping of MDE

• Moreover, to allow model formal analysis:
  - be **executable** to support model validation
  - be supported by a **set of tools** for model analyses
Integration

- Activities (1) and (2) are:
  - in principle unrelated
  - can be performed in parallel
  - two different notations for MDE and FMs can be used
In-the-loop Integration

Best results by \textit{in-the-loop integration} approach:
- MDE tech. and FM notation are \textit{the same} in both activities
- Act.1 proceeds Act.2
Instantiating the *In-The-Loop*

Eclipse Modeling Framework as MDE framework
- based on an open-source Eclipse framework
- unifies the three most important technologies
  - Java, XML, UML
- provides foundation for interoperability with EMF-based tools and applications

Abstract State machines as FM notation
- own the characteristics of a good helper lang.:
  - preciseness
  - abstraction
  - refinements
  - executability
  - metamodel-based definition
Abstract State Machines

• Extension of Finite State Machines
  - unstructured control states replaced by states comprising arbitrary complex data

• state is a multi-sorted first-order structure
  - in the sense of mathematical logic, that is, domains together with a number of functions

• transition rules to modify the state
  - structures can be viewed as algebras
    • basic transition rule: “if Condition then Updates”

• A computation of an ASM M:
  - a finite or infinite sequence $S_0, S_1, ..., S_n$ of states of $M$, where $S_0$ is an initial state and each $S_{n+1}$ is obtained from $S_n$ by firing simultaneously all of the transition rules which are enabled in $S_n$
Further characteristics of the ASMs

• Different but simple rule constructors to express control flow:
  - parallel (par) and sequential actions (seq);
  - iterations (while) and submachine invocations
  - non-determinism (choose) and synch. parallelism (forall)
  - synchronous/asynchronous multi-agents
Abstract State Machines

For a practical use, an ASM can be read as *pseudocode* on arbitrary data structures

- **ASM** = (header, body, main rule, initialization)
  - **header**: signature declaration, import/export of modules/components
  - **body**: domains/functions defs., rule decls., state invariant
  - **main rule**: the starting point of the machine program (it calls all the other ASM transition rules defined in the body)
  - **initialization**: specifies a set of initial states
    - an **initial state** defines an initial value for domains and functions declared in the signature

- **Executing** an ASM means executing its main rule starting from a specified initial state
Example of ASM

- **Tic-Tac-Toe game:**

A player plays against a computer. To win, three same symbols have to be placed in a row (horizontally, vertically, or diagonally). The game can end with a winning or in a draw, i.e. no spaces left on the board with none winning. If there is no winner after nine clicks, there is a tie.
Tic-Tac-Toe game specification

asm Tictactoe
signature:
//For representing a board
enum domain Skind = {CROSS | NOUGHT | EMPTY}
domain Square, Row subsetof Integer
controlled symbol: Square → Skind

//For managing the game
enum domain Finalres = {PLAYERX | PC | TIE}
enum domain Status = {TURNX | CHECKX | TURNPC | CHECKPC | GAMEOVER}
monitored playerX : Square // move of X
controlled status: Status
controlled whoWon: Finalres
derived noSquareLeft : Boolean
derived hasThreeOf: Prod(Row,Skind) → Boolean

//For PC strategies
controlled count: Integer
derived openingPhase: Boolean
controlled lastMoveX: Square
Tic-Tac-Toe game specification

main rule \texttt{r\_Main} =
\begin{verbatim}
par
    if status = TURNX then r\_movePlayerX[]
    if status = CHECKX then r\_checkForAWinner[CROSS]
    if status = TURNPC then r\_movePC[]
    if status = CHECKPC then r\_checkForAWinner[NOUGHT]
endpar
\end{verbatim}
Tic-Tac-Toe game specification

rule r_movePlayerX =
if symbol(playerX) = EMPTY
then par
  symbol(playerX):= CROSS
  status := CHECKX .....  endpar
else status := TURNX endif

rule r_movePC =
par
  r_tryStrategy[NOUGHT]
  status := CHECKPC .....  endpar
//A very naive player
rule r_try_strategy ($symbol in Skind) =
choose $s in Square with
  symbol($s)=EMPTY
  do symbol($s):= $symbol

rule r_checkForAWinner($symbol in Skind) =
//game over with a winner
if (exist $r in Row with hasThreeOf($r,$symbol))
then
  par
  status := GAMEOVER
  if $symbol = CROSS then whoWon:= PLAYERX
  else whoWon:= PC
  endpar
else if ( noSquareLeft )
then par
  status := GAMEOVER
  whoWon := TIE
  endpar
else
  if $symbol = CROSS then status:= TURNPC
  else status:= TURNX
Main reference: ASM book

• By Egon Boerger and Robert Staerk

• It also reports industrial and academic projects where ASMs have been applied
In-the-loop Integration with EMF and ASMs

Activity 1

Apply EMF to ASM

Apply ASM to EMF
ASM Language design by EMF/Ecore

- ASM language abstract syntax is defined by a MetaModel written in Ecore

- The AsmM (Abstract State Machines Metamodel) describes by an (object-oriented) model:
  - the vocabulary of concepts of the ASMs,
  - the relationships among concepts, and
  - how they may be combined to create models

- In a modular and bottom-up way:
ASM Language design by EMF/Ecore

115 classes
114 associations
150 OCL constraints
ASM  Tools development from the AsmM

- **Generated:** automatically derived from the meta-model by standard mappings to other technical spaces (XML, Java)

- **Based:** developed using artifacts (APIs, concrete syntaxes)
  - considerable amount of code is not generated

- **Integrated:** external and existing tools that are connected to the language artifacts
ASMETA tool-set

congrete syntax
eclipse plug-in
validator
simulator
verifier
artifacts
Test case generation
Asmeta tool-set

- Tools and material are available at the Asmeta website: asmeta.sourceforge.net
- GPLed/EPLed
In-the-loop Integration with EMF and ASMs

Activity 2

Apply EMF to EMF
Apply ASM to EMF
Activity 2: ASM for EMF

- We use ASM as helper language, with metamodel AsmM and well-defined semantic domain
- Different ways to define the building function $M$
Activity 2: ASM for EMF

- Techniques differ in the way a terminal model is mapped into an ASM
- One of these techniques: **semantic hooking**
  - An ASM $\Gamma_A$ is hooked to the language metamodel $A$
    - $\Gamma_A$ is an instance of AsmM and contains all data structures modeling elements of $A$ with their relationships, and all transition rules representing behavioral aspects of the language
    - $\Gamma_A$ does not contain initialization of functions and domains depending on a terminal model of $A$
  - The function $\iota_A: AsmM \times A \rightarrow AsmM$ adds the initialization part depending on the terminal model $m$
  - Formally, for each terminal model $m$ conforming to $A$:
    $$M(m) = \iota_A(\Gamma_A, m)$$
Applying the semantic framework to the Tic-Tac-Toe

Metamodel + OCL constraints

1. specify an ASM $\Gamma_{\text{Tic-Tac-Toe}}$ containing the signature and the semantics of the Tic-Tac-Toe metamodel in terms of ASM transition rules

2. add the function $\iota_{\text{Tic-Tac-Toe}}$ to initialize the ASM model

Get the already presented ASM model
Formal analysis: validation

- **interactive simulation**: use the simulator AsmetaS to interactively play Tic-Tac-Toe (player vs computer)

- **scenario-based simulation**: build scenarios

```
1 scenario winPC
2 load Tictactoe.asm
3 set playerX := 2;
4 step until status = TURNPC;
5 step until status = TURNX;
6 check symbol(2)=CROSS;
7 check symbol(5)=NOUGHT;
8 set playerX := 1;
9 step until status = TURNPC;
10 step until status = TURNX;
11 check symbol(1)=CROSS;
12 check symbol(3)=NOUGHT;
13 set playerX := 8;
14 step until status = GAMEOVER;
15 check symbol(7)=NOUGHT;
16 check whoWon = PC;
```
Formal analysis : properties verification

Use AsmetaSMV model checker to prove that:

- The specification fair and both player can win:
  //the player can win
  CTLSPEC : EF(whoWon=PLAYER)
  //the computer can win
  CTLSPEC : EF(whoWon=PC)
  //the match can terminate tie
  CTLSPEC : EF(whoWon=TIE)
- The match always finishes:
  CTLSPEC : AF((status = GAMEOVER))
  (proved false and a counter example was provided)
More applications of the formal framework

• Simple examples:
  - Semantics of FSM and Petri Nets from their metamodels

• More complex examples:
  - semantics of the AVALLA language, a domain-specific modeling language for scenario-based validation of ASM models
  - Semantics of the SystemC UML profile, a modeling language for embedded systems developed in collaboration with STMicroelectronics

• Recently: SCA-ASM Language
  - a modeling language, based on the standard SCA and on the distributed multi-agent ASMs, for formal and executable description of services internal behavior, services orchestration and interactions
Conclusions

- On our experience in developing ASMETA, we believe a **FM can gain benefits from the MDE automation** for itself and toward the integration with other formal techniques/tools.

- A means for specifying **rigorous semantics of metamodels** is a necessary step to develop formal analysis techniques/tools in the model-driven context.
Conclusions

- We **closed-the-loop** by defining the semantics of the AsmM metamodel by using the ASM/EMF-based semantic framework

- I will be @FAU till Sept. 3 and open to discussion