Towards Model Checking VDM

Hsin-Hung Lin

School of Information Science and Electrical Engineering
Kyushu University

Kaken(S) Research Meeting
5 March 2014
Nishijin Plaza
Kyushu University
Outline

1 Introduction

2 Problem Definition

3 Approach

4 Summary
Outline

1. Introduction
2. Problem Definition
3. Approach
4. Summary
Software development is divided to several phases with corresponding documents/products:

- **Requirement**: Natural Language + Domain Knowledge + …
- **Specification**: Formal or Informal Specification Languages (VDM-SL or VDM++ or Z or B or …)
- **Design**: UML or SysML or …
- **Implementation**: C or Java or …
Software Development in General

Documents/Products below are guaranteed by validation with above ones

- There may be feedbacks/revisions from below to above

- Requirement: Natural Language + Domain Knowledge + …

- Specification: Formal or Informal Specification Languages (VDM-SL or VDM++ or Z or B or …)

- Design: UML or SysML or …

- Implementation: C or Java or …
A document/product itself should be verified its own correctness

- Could be formal or informal, or semi-formal

**Diagram:**

- Requirement: Natural Language + Domain Knowledge + …
- Specification: Formal or Informal Specification Languages (VDM-SL or VDM++ or Z or B or …)
- Design: UML or SysML or …
- Implementation: C or Java or …
Specification Centric Software Development using VDM

Basic Idea

- Inspired by FeliCa card development
- Specification written in VDM as the fix-point in software development
- If validated and verified, VDM specification can be used as test/validate oracle of designs and implementations

Three Main Issues

- Write VDM specification according to requirements
- Correctness of VDM specification (current focus!)
  - Verification/Validation with requirements
- Software testing based on VDM specification
  - Test case generation
Specification Centric Software Development using VDM

Basic Idea

- Inspired by FeliCa card development
- Specification written in VDM as the fix-point in software development
- If validated and verified, VDM specification can be used as test/validate oracle of designs and implementations

Three Main Issues

- Write VDM specification according to requirements
- Correctness of VDM specification (current focus!)
  - Verification/Validation with requirements
- Software testing based on VDM specification
  - Test case generation
Outline

1. Introduction
2. Problem Definition
3. Approach
4. Summary
Vienna Development Method (VDM)

- Model-Based Specification Language
  - Proposed in 70's by IBM Vienna laboratory
  - Types, Variables (state), Operations, Functions
- Pre/Post conditions of Operations + Invariants of Variables
  - Core of VDM
  - Similar to OCL in UML
  - For generating proof obligations (Theorem proving!)
- Executable explicit specification
  - By interpreter: Overture(VDMJ), VDM tools
  - Verification/Validation by testing

Some options to verify/validate a VDM specification

- Theorem proving: Discharging proof obligations
  - Pros: proofs give best guarantees
  - Cons: Depend on pre/post-conditions, not automated, ...

- Testing
  - Pros: Easy to conduct, basic guarantees
  - Cons: coverage, difficult to be exhaustive, ...

- Model Checking (current focus!)
  - Pros: exhaustive check (with limitation), automated, good guarantees
  - Cons: state explosion
Existing work: “Translating VDM to Alloy”

- Target: subset of VDM-SL
- Semantic preserving translation
- Focus on translation instead of checking properties
- Pros: intuitive
- Cons: what to check is not clear (difficulty of expressing properties)

Our Basic Idea

- Think again from the origin of model checking

Goal

- Define the model checking problem of VDM specifications and implement its solution which can
  - Check post-conditions and invariants
  - Check temporal properties (safety, liveness)
Outline

1. Introduction
2. Problem Definition
3. Approach
4. Summary
Overview of Approach

- No direct access to VDM specification but through interpreter
- No need to worry about handling semantics of VDM
- Model checker only sees interfaces of VDM specification
A VDM model is represented as a 4-tuple:

\[ \mathcal{M} = (\text{Types}, \text{Var}, \text{Fun}, \text{Ope}) \]

where

- **Types** is the finite set of types.
- **Var** is the finite set of variables (state);
  - \( \text{type}(v), \text{type}(v) \in \text{Types} \), denotes the type of \( v \in \text{Var} \).
  - \( \text{inv}(v) \) is the invariant of \( v \).
- **Fun** is the finite set of functions;
- **Ope** is the finite set of operations;
  - \( I(op) \) and \( O(op) \) denotes the set of inputs and outputs of \( op \in \text{Ope} \) respectively;
  - \( \forall \text{in} \in I(op), \text{type}(\text{in}) \in \text{Types} \);
  - \( \forall \text{out} \in O(op), \text{type}(\text{out}) \in \text{Types} \);
  - \( \text{pre}(op) \) and \( \text{post}(op) \) represent the precondition and postcondition of \( op \in \text{Ope} \) respectively.
For a VDM model $\mathcal{M} = (\text{Types}, \text{Var}, \text{Fun}, \text{Ope})$, the interpreter provides functions $\text{dom}$, $\text{val}$, and $\text{eval}$

- $\text{dom}(t)$ denotes the finite domain of $t \in \text{Types}_\mathcal{M}$, i.e. the finite set of all values of $t$.
- $\text{val}(v)$ returns a value of $v \in \text{Var}_\mathcal{M}$, $\text{val}(v) \in \text{dom}(\text{type}(v))$.
- $\text{eval}(\text{op}) \subseteq \text{Dom}(I(\text{op})) \times \text{Dom}(\text{Var}_\mathcal{M}) \times \text{Dom}(\text{Var}_\mathcal{M}) \times \text{Dom}(O(\text{op}))$
  
  - $\text{eval}(\text{pre}(\text{op})) \subseteq \text{Dom}(I(\text{op})) \times \text{Dom}(\text{Var}_\mathcal{M}) \times \{T, F\}$ evaluates the precondition of $\text{op} \in \text{Ope}_\mathcal{M}$.
  
  - $\text{eval}(\text{post}(\text{op})) \subseteq \text{Dom}(I(\text{op})) \times \text{Dom}(\text{Var}_\mathcal{M}) \times \text{Dom}(\text{Var}_\mathcal{M}) \times \text{Dom}(O(\text{op})) \times \{T, F\}$ evaluates the postcondition of $\text{op} \in \text{Ope}_\mathcal{M}$.  


The behavior derived from a VDM model $\mathcal{M} = (\text{Types}_\mathcal{M}, \text{Var}_\mathcal{M}, \text{Fun}_\mathcal{M}, \text{Ope}_\mathcal{M})$ is represented as an extended automaton $EA^{VDM}_\mathcal{V}$:

$P = (S, s_0, \mathcal{E}, \mathcal{T}, \mathcal{V}, A)$, where

- $S$ is the finite set of states;
- $s_0 \in S$ is the initial state;
- $\mathcal{E}$ is the finite set of events/actions;
  - $e = (op, val(I(op)))$, where $e \in \mathcal{E}$, $op \in \text{Ope}_\mathcal{M}$;
- $\mathcal{V} = V_\mathcal{M}$ is the finite set of variables.
- $A \subseteq \text{Fun}_\mathcal{M}$ is the set of propositions.
  - For a proposition $a \in A$, $I(a) = \phi$ and $O(a) = \{p\}$ where $p$ is of type boolean.
- $\mathcal{T} \subseteq S \times \mathcal{E} \times S$ is the set of transition relations;
  - For a transition $t \in \mathcal{T}$, $t = (s, e, s')$ where $s, s' \in S$ and $e \in \mathcal{E}$, $t$ is valid if and only if $\text{eval}(\text{pre}(op)) == \text{True} \land \text{eval}(\text{post}(op)) == \text{True} \land \forall v \in \mathcal{V}, \text{eval}(\text{inv}(v)) == \text{True}$ for $s$ and $s'$.
Definition (Configurations of $EA^{VDM}$)

Given an extended automaton $EA^{VDM}: P = (S, s_0, \mathcal{E}, T, \mathcal{V}, A)$ derived from a VDM model $\mathcal{M} = (Types_{\mathcal{M}}, Var_{\mathcal{M}}, Fun_{\mathcal{M}}, Ope_{\mathcal{M}})$ and the VDM interpreter,

- The finite set of configurations of $P$ is denoted as $C_P \subseteq \text{Dom}(Var_{\mathcal{M}}) \times S$
- The finite set of initial configurations of $P$ is denoted as $C^0_P \subseteq \text{Dom}(Var_{\mathcal{M}}) \times \{s_0\}$
- $C^0_P \subset C_P$
Definition (Traces of \( EA^{VDM} \))

Given the configurations \( C_P \) of an extended automaton \( EA^{VDM} \):
\( P = (S, s_0, \mathcal{E}, \mathcal{T}, \mathcal{V}, \mathcal{A}) \) derived from a VDM model
\( M = (\text{Types}_M, \text{Var}_M, \text{Fun}_M, \text{Ope}_M) \) and the VDM interpreter,
\( \pi = c_0 c_1 c_2 \ldots \) denotes a trace of \( P \), where

- \( c_0 \in C_P^0 \)
- \( c_i \in C_P, i \in \mathbb{N}^0 \)
- For \( c_i = (\text{val}_i(V_M), s_i) \) and \( c_{i+1} = (\text{val}_{i+1}(V_M), s_{i+1}) \), \( \exists t \in \mathcal{T}, \ t = (s_i, e, s_{i+1}), e \in \mathcal{E} \) where \( e = (op, \text{val}(I(op))) \), \( op \in \text{Op}_M \),
  \( \text{eval}(op) = (\ \text{val}(I(op)), \text{val}_i(V_M), \text{val}_{i+1}(V_M), \text{val}(O(op))) \).
Definition (LTL semantics of $EA^{VDM}$)

Given a trace $\pi = c_0c_1c_2 \ldots$ of an extended automaton $EA^{VDM}$: $\mathcal{P} = (S, s_0, \mathcal{E}, T, \mathcal{V}, \mathcal{A})$ derived from a VDM model $\mathcal{M} = (\text{Types}_\mathcal{M}, \text{Var}_\mathcal{M}, \text{Fun}_\mathcal{M}, \text{Ope}_\mathcal{M})$ and the VDM interpreter. Let $p \in \mathcal{A}$ and $\phi, \varphi$ are LTL formulas constructed from propositions $a \in \mathcal{A}$,

- $\pi \models p \iff \text{eval}(p) = (\text{val}_0(\mathcal{V}_\mathcal{M}), T)$
- $\pi \models \neg \phi \iff \pi \notmodels \phi$
- $\pi \models \phi \land \varphi \iff \pi \models \phi \land \pi \models \varphi$
- $\pi \models \phi \lor \varphi \iff \pi \models \phi \lor \pi \models \varphi$
- $\pi \models \Box \phi \iff \forall i \geq 0. \pi^i \models \phi$
- $\pi \models \Diamond \phi \iff \exists i \geq 0. \pi^i \models \phi$
- $\pi \models \phi \ U \varphi \iff \exists i \geq 0. \pi \models \varphi \land \forall 0 \leq j < i. \pi^j \models \phi$
Plan of implementation

- SPIN (use of embed C) + VDMJ (interpreter of Overture)
  - Need C functions to call operations and functions using interfaces of VDMJ
  - Also need operations for retrieving and restoring state/instance variables
- Fast abstract example: A simple elevator system (3 floors)
Example (Simple Elevator)

\[ \mathcal{M} = (\text{Types}, \text{Var}, \text{Fun}, \text{Ope}) \]

- **Types** = \{Floor, Cage, System\}
- **Var** = \{floors, cage, system\}
- **Fun** = \{P1, P2\}
- **Ope** = \{PushOpenButton, PushCloseButton, PushFloorButton, PushCallButton, PushFloorButton, PushCallButton, OpenDoor, CloseDoor, MoveCage\}
Example (Simple Elevator)

Fast abstract of model checking the system

- Put the automaton model in a model checker (ex: SPIN+VDMJ)
- Check without temporal properties
  - No counterexample: all post-conditions and invariants are satisfied
- Check temporal properties
  - Let P1: door at floor 1 is open
  - Let P2: door at floor 2 is open
  - Safety: $[]! (P1 \& \& P2)$

- Let P1: floor button of floor 3 is pushed (request)
- Let P2: cage at floor 3 and the door at floor 3 is open (response)
- Liveness: $[] (P1 \rightarrow <> P2)$
Outline

1 Introduction

2 Problem Definition

3 Approach

4 Summary
An approach of model checking VDM specifications

- Current progress
  - Definition of model checking problem
  - Indirect way
  - Utilize the interpreter of VDM

- Implementing solution (plan)
  - SPIN(embed C) + VDMJ

- Fast abstraction of the usage of model checking VDM
  - Simple elevator example
Thank You.

Q & A