The Logic of RAISE Specification Language

Qinglei Zhang

Department of Computing and Software, McMaster University
Canada L8S 4L7, Hamilton, Ontario

March 17, 2009
RAISE is a product providing facilities for the industrial use of Formal Methods in the development of software systems.

- RAISE was developed during the year 1985-1995
- The RAISE language and tools focus on supporting the specification, design and implementation stages of the software development process.
- RAISE supports development all the way to final compilable code. The entire development from specification through to implementation will be formally recorded when using RAISE.
Definition (Formal Methods)

methods with a well-defined syntactical and semantical level where both of them are based on mathematical theories (languages, set theory, logic, algebra). F.M. Include:

1. **Formal systems:** formal language with 1) well-defined syntax, 2) well defined semantical and 3) proof systems

2. **Development technique:** Implementation produced from specification; Application of development steps; refinements process.

3. **Verification technique:** verify whether the implementation satisfieds the specification; verify the soundness of each development step.
RAISE: Rigorous Approach to Industrial Software Engineering

RAISE consists of:

1. Specification Language (RSL), which is a powerful specification and design language
2. A comprehensive development method: stepwise refinement using the invent and verify paradigm.
3. Extensive tool support for producing specifications, theories and proofs:
editing, performing justifications translating into imperative languages document support
RSL: RAISE Specification Language

RSL specifications: mathematical models of software systems.

- It is a wide-spectrum specification language: may be used to express high-level, abstract specifications, as well as low-level designs. The structuring facilities of RSL support decomposition and re-use.
- RAISE builds upon ideas reflected in a number of other formal methods and languages.
  - algebraic feature: e.g. OBJ, Clear, ASL, ACT ONE
  - model-oriented feature: e.g. VDM and Z
  - concurrency features: e.g. ASL
  - modularity features: e.g. ML
A basic RSL specification is called a class expression and consists of declarations of:

- types: abstract sort types (algebraic specification) or model-oriented way
- values: defined in a signature-axiom style (algebraic specification) or explicit styles
- Functions: variables, channels, axioms: may be imperative or applicative
Definedness of RSL

Problem: what to do about problematic expressions:

1. such expressions cannot be avoided;
2. there is a variety of schemes available to deal with such expressions:

choice of solution:

1. affecte by the conveniently translate at least the constructes likely to appear in such detailed specifications into a programming language;
2. The design of RSL is as regular as possible.
Definedness of RSL

RSL allows in its logic for expressions that might not denote values in their types, i.e., there are some expressions whose definedness is not specified. *definedness of expressions is a concept needed in the proof theory.*

- only defined expressions have values
- three basic undefined expressions in
  1. *chaos*(equivalent to *while true do skip end*)
  2. *stop*(represents deadlock)
  3. *swap*(the interval choice over the empty set)
Convergence of RSL

Definition (Convergence)

*defined and having a unique value*

- RSL includes concurrency, and so includes the notion of internal choice.
- Needed for the hold of commutativity and ”excluded middle”
Connective of RSL

Connective: define in a ”conditional logic”. e.g. $A \land B \equiv$ if $A$ then $B$ else false end

- if $A$ then $B$ else $C$ end, means:
  1. If $A$ is undefined, then the expression is equivalent to $A$.
  2. otherwise, if $A$ is nondeterministic the expression is equivalent to $B \sqcap C$
  3. otherwise, if $A$ is true the expression is equivalent to $B$, and if $A$ is false then the expression is equivalent to $C$

- lose the unconditional commutativity of $\land$ and $\lor$, the implementability of the logic was overriding concern.
Quantifiers, Functions and Axioms of RSL

- Quantify over values in the appropriate type
- Functions only applied to values in their domain
- Axioms may be declared. In addition, all value declaration are short for value signatures plus axioms.
Applicative RSL

Definition (Equivalence $\equiv$ (equivalence or strong equivalence) has the mathematical properties of a congruence, that's it is an equivalence relation)

1. A defined expression is never equivalent to an undefined one.
2. Equivalence always returns true or false.

Definition (Equality is just like that of any other infix operator. $A = B$ means,)

1. If either of $A$ or $B$ is undefined then so is the equality. If it is undefined so is the whole expression.
2. If either of them is nondeterministic, then so is the equality.
3. Otherwise, the equality is the same as $\equiv$. 
- Expressions that can have effects, i.e. can read or write variables, or input from or output to channels.
- Problem: What should be the effect on the variable of parallel evaluation.
- Imperative specification: depend very heavily on evaluation order.
Imperative RSL

Two expressions to be equivalent: have equivalent effects as well as equivalent results.

**Definition (Equivalence: Logical equivalence)**
- *There is no actual evaluation*
- *To obtain a congruence relations, introduce an extra connective “always”, □p means “p is true in any state”, i.e. regardless of the contents of variables.*

**Definition (Equality: evaluated left to right.)**
- *If the expression on the left has effects, these can affect the result of the expression on the right.*
- *It consistent with most programming languages.*
Definition (Evaluation Order)

We need to be clear about the evaluation order of all expressions.

E.g. the evaluation rule for if-expressions: if A then B else C end ≡ let x=A in if x then B else C end end

where the identifier x is chosen so as not to be free in B or C
Reasoning Style

- base on equivalence: imperative descriptions as well as applicative descriptions.
- based on Hoare logic: weakest preconditions for sequential imperative descriptions and perhaps temporal logic for concurrent descriptions.
Refinement in RSL

- RSL is modular language, refinement is aimed in particular at allowing substitution.
- RSL is required to be monotonic with respect to module composition.

Definition (refinement in RSL is that class expression $B'$ refines class expression $B$ provided:)

1. The signature of $B'$ include that of $B$
2. All the properties of $B$ hold in $B'$. 
Axiomatic Semantics

- Proof theory is to provide an axiomatic semantics that defines the meaning of RSL.
- The purpose of proof
  - provide formation rules for determining whether a specification is well-formed
  - proof rules for reasoning about specification, e.g. whether two RSL terms are equivalent or whether an RSL specification is refinement of another RSL specification
Axiomatic Semantics

The Collection of Inference Rules

- Formation Rules
- Context-independent Equivalence Rules: two terms are equivalent in all respects.
- Context-dependent Equivalence Rules: a given context two terms are equivalent in the weaker sense that they have the same meaning
- Refinement Rules
- Auxiliar Rules: used in the premises of other rules.
Soundness: The denotational semantics provides evidence of the existence of a model satisfying the proof rules. [3]
Completeness: Defines a collection of “normal forms” and adds rules to show how other forms may be made normal.
The RSL Proof System:

- sound but incompleated (sacrificing completeness to obtain more ease of proof)
- The proof rules are intended for application by a toll, the RAISE justification editor
- The language for expression proof rules is a small extension of RSL

Definition Versus Proof

- the defining rules need to be concerned with well-formedness; the proof rules can assume terms are well formed
- The smaller a set of defining rules we have, the more easily can we show it to be sound, the larger the set of rules for proof we have the easier proofs will be.
A Simple Version of an applicative model-oriented specification

- Requirements: Provide some functions to allow the harbor master to control the movement of ships in and out of the harbor.

Verify the Specification

- Proof is ideal method
  - Time consuming and requires expertness
  - Currently used for very critical Software or in hardware design
  - Support tools: justification editor (eden), Translation (into PVS)

- Other ways to Verify Specifications
  - Checking or testing to obtain more assurance
  - Progressively cheaper to use, but also progressively less assurance
  - Several alternatives: Model Checking, Testing, Inspection
Reference

1. Logics of Specification Languages: the Logic of the RAISE Specification Language
5. RAISE methods: http://www.iist.unu.edu/raise/