Understanding Formal Specifications

Leo Freitas and Zoë Stephenson
University of York
HISE Seminar
02 June 2009
The role of specification

- depends on what your concerns are
- common concern is functional correctness
  - partial: artefact satisfies a contract.
  - total: partial with termination guarantees.
- other important concerns, e.g.,
  - fault-tolerance / robustness
  - exception / deadlock freedom
  - error detection / correction / precision
  - particular standard compliance
- functional correctness is often harder to establish
The role of specification

- **partial correctness: contract**
  - contract conditions ⇒ deliverable guarantee.
  - a.k.a., precondition ⇒ postcondition.

- **other important aspects**
  - invariants: system-wide guarantees.
  - variants: termination conditions.

*Let’s see an example*

Given a non-empty set of integers, calculate their minimum.
The role of specification

- why are such concerns important?
- could you judge somebody’s choice of formalism?
- what is the specification / contract?
- does the implementation satisfy the contract?

Industry examples

- Praxis model of software contract (SPARK)
- QinetiQ’s RSG implementation platform (Ada)
- Airbus source code analysis (Ashtray, Esterel, C)
- Microsoft device driver tools (C, C#, C++)
Minimum example under the bonnet:

given that:

\[ x, y, z, r \in \mathbb{Z} \]
\[ P \equiv (x < y) \land (x < z) \]
\[ Q \equiv (y < x) \land (y < z) \]

Our program can be formally expressed as

\[ r = x \triangleleft P \triangleright (r = y \triangleleft Q \triangleright r = z) \]

Various programming calculi exist for more complicated constructs
\[ r = x \triangleleft P \triangleright (r = y \triangleleft Q \triangleright r = z) \]

\[ \overset{\text{def. of cond.}}{=} (P \land r = x) \lor (\neg P \land (r = y \triangleleft Q \triangleright r = z)) \]

\[ \overset{\text{def. of cond.}}{=} (P \land r = x) \lor (\neg P \land ((Q \land r = y) \lor (\neg Q \land r = z))) \]

\[ \overset{\text{case 1: } x \text{ is min.}}{=} (P \land r = x) \lor (\neg P \land Q \land r = y) \lor (\neg P \land \neg Q \land r = z) \]

\[ \overset{\text{case 2: } y \text{ is min.}}{=} \]

\[ \overset{\text{case 3: } z \text{ is min.}}{=} \]
Case 1: \( x \) is min.

\[(x < y \land x < z) \land r = x\]

Case 2: \( y \) is min.

\[((y < z \leq x) \lor (y < x \land y < z)) \land r = y\]

Case 3: \( z \) is min.

\[((z \leq x \leq y) \lor (z \leq y \leq x) \lor (x = y)) \land r = z\]
Tool support is paramount

- **specification detection:** Daikon, Pex, Prefix, etc.
  - find invariants given a “good” test-set

- **static specification analysis:** SPARK, Spec#, JML, etc.
  - guarantees impl. satisfies the spec. (e.g., wp-calculus)
  - may not avoid run-time errors (e.g., overflow, deadlock)

- **model checkers:** Spin, Nu-SMV, FDR, Alloy, etc.
  - specification debuggers: are we asking the right questions?
  - errors are traceable counter-examples

- **theorem provers:** Isabelle, HOL, PVS, CoQ, ACL2, Z/Eves, etc.
  - specification verifiers: strongest possible guarantees
  - harder to use: requires user interaction / expertise
Understanding Formal Specifications
Understanding Formal Specifications

[Diagram of a network of buildings with distances labeled]

- Computer Science
- Library
- Vanbrugh
- Langwith
- Central Hall
- Physics & Electronics
- Alcuin
- Chemistry
- Derwent
- Heslington Hall
- SBD
- Concert Hall
- Biology
- Goodricke
- Wentworth
- James

Distances:
- Computer Science to Alcuin: 21
- Library to Alcuin: 15
- Vanbrugh to Alcuin: 11
- Langwith to Alcuin: 15
- Central Hall to Alcuin: 16
- Physics & Electronics to Alcuin: 38
- SBD to Concert Hall: 10
- Concert Hall to Library: 19
- Concert Hall to Vanbrugh: 19
- Concert Hall to Central Hall: 26
- Concert Hall to Physics & Electronics: 40
- Biology to Concert Hall: 15
- Biology to Vanbrugh: 17
- Biology to Goodricke: 20
- Biology to Wentworth: 19
- Biology to James: 18
- Goodricke to Vanbrugh: 7
- Goodricke to Central Hall: 11
- Goodricke to Physics & Electronics: 9
- Wentworth to Goodricke: 19
- Wentworth to James: 15
- Wentworth to Physics & Electronics: 21
- James to Vanbrugh: 19
- James to Physics & Electronics: 21
- James to SBD: 13
- SBD to Biology: 15
- SBD to Concert Hall: 13
- SBD to Goodricke: 19
- SBD to James: 18

Understanding Formal Specifications – p. 6/16
Graph Definitions

- A bunch of things — nodes, uniquely identified
- Another bunch of things — edges (arcs)
- A rule — each edge connects two nodes
- Edge labels (numeric labels are often called weights)
- Edge endpoint labels
- Directed edges
- Different node types
- Different edge types
- Nested nodes
Graph Maths

\[ G = (V, L, E) \]

e.g.

\[ G = (\{a, b, c\}, \{x, y\}, \{(a, b, x), (b, c, y), (b, a, y), (b, a, x), (c, a, x)\}) \]

Can also use a matrix representation:

\[
G = \begin{pmatrix}
\{a, b, c\}, \{x, y\} & \{\}, \{x\}, \{\}, \\
\{a, b, c\}, \{x, y\} & \{x, y\}, \{\}, \{y\}, \\
\{a, b, c\}, \{x, y\} & \{x\}, \{\}, \{\}
\end{pmatrix}
\]
Graphs — Motivation

- Recognising common structural themes
- Developing general-purpose algorithms to apply in many situations
  - e.g. shortest-path, reachability, ...
- Recognising types of graph with particular properties
  - Acyclic graph
  - Bipartite graph
  - ...
- Using appropriate algorithms and other graph-theoretic results to write efficient programs
Almost all applications use graphs:

- **Compilers and static program analysers** — call graph, state space graph, module dependency graph
- **UML and SysML** — class diagrams, activity diagrams, …
- **Transport** — route management, rail network
- **Mind-mapping**
- …
Understanding Formal Specifications

Diagram:

- Stowed
- Deployed

Arrows:
- reverse command
- forward command
State Machine Definitions

- A bunch of things — states, uniquely identified
- Another bunch of things — transitions
- A rule — each transition connects two states
- One state is the initial state
- A subset of the states can be final states

- Current state
- Each transition can have a guard label
- From a given state, transitions with guards satisfied are enabled
- If any transitions are enabled, one is taken
- When a transition is taken, it updates the current state
State Machine Maths

\[ M = (S, s_0, F, T) \]

- states
- initial state
- final states
- transitions
State Machines — Motivation

- Recognising common structural themes
- Developing general-purpose algorithms to apply in many situations
  - e.g. reachable state, deadlocked state, . . .
- Recognising types of state machine with particular properties
  - Algebraic loops
  - Race conditions
  - . . .
- Using appropriate types of state machine to help intuition, model behaviour and auto-generated code align as much as possible
State Machines — Applications

Lots of things use state machines:

- Text parsing
- UML/SysML, Matlab/Simulink/Stateflow
- Communication protocols
- ...

But see also: Markov model, Bayesian network, ...

Understanding Formal Specifications – p. 16/16