Axiomatic Specification

Al-Said, Apcar, Jerejian
• Axioms:
  • Wffs that can be written down without any reference to any other Wffs.
  • Wffs that are stipulated as unproved premises for the proof of other wffs inside a formal system.

• Rules of Inference:
  Rules that allow us to produce Wffs as immediate consequences of other Wffs.
Mathematics

Group Axioms:

\[(x \ast y) \ast z = x \ast (y \ast z)\]

\[x \ast e = x\]

\[e \ast x = x\]

\[x \ast x^{-1} = e\]

Theorem:

\[x^{-1} \ast x = e\]

Applications:
Proof Checking
Proof Generation
Database Systems

Parent Database:

<table>
<thead>
<tr>
<th>parent</th>
<th>grandparent</th>
</tr>
</thead>
<tbody>
<tr>
<td>art</td>
<td>cal</td>
</tr>
<tr>
<td>amy</td>
<td>coe</td>
</tr>
<tr>
<td>bob</td>
<td>cal</td>
</tr>
<tr>
<td>bob</td>
<td>coe</td>
</tr>
</tbody>
</table>

Database in Sentential Form:

\begin{align*}
\text{parent}(\text{art}, \text{bob}) & \quad \text{grandparent}(\text{art}, \text{cal}) \\
\text{parent}(\text{art}, \text{bea}) & \quad \text{grandparent}(\text{art}, \text{coe}) \\
\text{parent}(\text{bob}, \text{cal}) & \quad \text{grandparent}(\text{amy}, \text{cal}) \\
\text{parent}(\text{bea}, \text{coe}) & \quad \text{grandparent}(\text{amy}, \text{coe}) \\
\end{align*}

Constraints:

\begin{align*}
\text{parent}(x, x) \\
\text{parent}(x, y) & \Rightarrow \neg \text{parent}(y, x) \\
\text{parent}(x, y) \land \text{parent}(y, z) & \Rightarrow \text{grandparent}(x, z)
\end{align*}

Applications:

Query Planning
Query Folding
Integrity Constraints

http://meta2.stanford.edu/classes/cs157/
Digital Circuits

Circuit:

Behavior:

\[ o \Leftrightarrow (x \land \neg y) \lor (\neg x \land y) \]
\[ a \Leftrightarrow z \land o \]
\[ b \Leftrightarrow x \land y \]
\[ s \Leftrightarrow (o \land \neg z) \lor (\neg o \land z) \]
\[ c \Leftrightarrow a \lor b \]

Applications:
Simulation
Configuration
Diagnosis
Test Generation

http://meta2.stanford.edu/classes/cs157/
Software Analysis

Program:

\[ L \rightarrow \text{sorter} \rightarrow \text{sort}(L) \]

Specification:

\[ i < j \Rightarrow \text{elt}(\text{sort}(l), i) \leq \text{elt}(\text{sort}(l), j) \]

Applications:
- Verification
- Partial Evaluation
- Complexity Analysis
- Proofs of Termination

http://meta2.stanford.edu/classes/cs157/
HOW To Prove Program Correctness:

• Use mathematical logic techniques to proof if a computer program carries out its intended function.

• Using deductive reasoning, we can find out all the consequences of a computer program execution.

• Deductive reasoning: Applying rule of inference to sets of axioms.
Rules of Inference (Subset):

- **Modus Ponens or Implication-Elimination:**
  \[(A \Rightarrow B, A) \vdash B\]

- **And-Elimination:**
  \[(A_1 \text{ and } A_2 \text{ and } \ldots \text{ An}) \vdash A_i\]

- ........
• Syntax of FOL
  • Sentence -> Atomic Sentence
    | Sentence Connective Sentence
    | Quantifier Variable,... Sentence
    | NOT Sentence
    | (Sentence)
  • Atomic Sentence -> Predicate(Term, ...) | Term = Term
  • Term -> Constant
    | Variable
    | Function(Term,...)
  • Connective -> AND | OR | => | <=>
  • Quantifier -> FORALL | EXISTS
  • Constant -> A | X1 | John | ...
  • Variable -> a | x | s ...
  • Predicate -> Before | HasColor | Raining | ...
  • Function -> Smelly | LeftLegOf | Plus | ...
• Assertions:
  • logical expressions

• a true-false statements about the state of the program.

• Typically, an equality or inequality relating the values of various identifiers in the code.
• **Main Concept:**

• The results of a program or part of a program depends on the values taken by the variables before that program is initiated.

  \[ P \ {\{Q\}} \ R \ \text{P: precondition (constraint) } , \ Q \ \text{program, R post condition or result} \]

• We need axioms and rules that relate post conditions to preconditions and vice versa.
• How:

• Axioms and Rules for for every kind of statement in the programming language under consideration.

OR

• A small number of primitive statements, then
• Define other statements in terms of those primitives
Program Execution Axioms:

Assignment Axiom:

D0:

\[ \vdash P0 \{ \chi := f \} \quad P \quad \text{Axiom Schema} \]
• Example:

• What is the Pre condition for \{\text{sum} > 1\} to be true, after executing "\text{sum} := 2x + 1"?

\{\text{sum} > 1\} \ [\text{sum} := 2x + 1] \ \{\text{sum} > 1\}

and

\{\text{sum} > 1\}[2x + 1 / \text{sum}]
  = \{2x + 1 > 1\}
  = \{2x > 0\}
  = \{x > 0\}

➢ so if \{x > 0\} is true and we execute \text{sum} := 2x + 1, then \{\text{sum} > 1\} is true.
• Rules of Consequence:

D1:

if \( \vdash P \{Q\} R \land \vdash R \rightarrow S \) then \( \vdash P\{Q\}S \)

if \( \vdash P \{Q\} R \land \vdash S \rightarrow P \) then \( \vdash S\{Q\}R \)

• Rule of Composition:

D2:

if \( \vdash P \{Q1\}R1 \land \vdash R1 \{Q2\}R \) then \( \vdash P\{(Q1;Q2)\}R \)
• Rule of Iteration

D3:

\[ \text{if } \vdash P \land P \{S\}P \text{ then } \vdash P \{\text{while } B \text{ do } S\} \ 
eg B \land P \]
Limitations of above Axioms:

- Simple programs only.
- Assume no side of effects of expressions and conditions evaluation.
- Do not detect improper program termination
- Do not cover non-integer arithmetic
• Program Correctness:
  
  • Possible when both PL and H/W are implemented based on axioms that describe their logical properties.

• Programs proving will solve the 3 most pressing problems:
  • Reliability
  • Documentation
  • Compatibility
Neochiron (subset)

Fα: α Fragment;  
Tα: α Transport  
IPα: α Input Portal;  
OPα: α Output Portal  
Jα: α Jumper;  
Sα: α State;  
C: Connect  
Mα: α Method;  

∀x IP(Fx) → ¬OP(Fx)
∀x OP(Fx) → ¬IP(Fx)
∀x IP(Tx) → ¬OP(Tx)
∀x OP(Tx) → ¬IP(Tx)
∀xyz Fx ∧ Ty ∧ Jz → C(Jz, OP(Fx), IP(Ty))
∀xyz Fx ∧ Ty ∧ Jz → C(Jz, IP(Fx), OP(Ty))
∀xz Fx ∧ Jz → ¬C(Jz, OP(Fx), IP(Fx))
∀xyz Fx ∧ Fy ∧ Jz → ¬C(Jz, OP(Fx), IP(Fy))
\[\forall xyz \quad Tx \land Ty \land Jz \rightarrow \neg C(Jz, \text{OP}(Tx), \text{IP}(Ty))\]
\[\forall x \quad Sx \in \{\text{Run, Halt, Snooz, Suspend}\}\]
\[\forall x \quad Mx \in \{\text{Start, Suspend, Halt}\}\]
Formal Methods

- Produce models, define properties of system at several levels of abstraction
- Formal methods can be used at each level
- Aspects of a system that can be specified by formal methods
  - functionality
  - safety (e.g. unsafe states will not arise)
  - security
Requirements Analysis and Specification

- Draw out, clarify and document requirements of a computation system
- Produce corresponding functional specifications
- Complete and precise statements of requirements and constraints on system functions
Using Axiomatic Specification in Formal Methods

- Derive programs from formal functional specifications in first order logic and algebraic languages
- Prove program correctness with respect to such formal specifications
Formal Representation

- Provide solution to ambiguity problems
- Formal proof and analysis of consistency and completeness
- Formal functional specification in first order logic using algebraic formal specification languages may be considered as the base of software formal development
- Use of a formal language (e.g. Z ("Zed"))
Formal Representation (2)

- Proof begins with set of axioms (statements postulated to be true).
- Inference rules - deriving other formula from axioms (premises to consequent).
- A *proof* consists of a sequence of formulae in the language in which each formula is either an axiom or derivable by an inference rule from previous formulae in the sequence.
Defining Axiomatic Specification

- The *Axiomatic Specification* is a formal specification defining the semantics of functions of objects by a description of the relations between different objects and functions. The description is made by axioms (predicate-logical formula).

  - Baader, 1990
Heterogeneous Typing for Software Architectures

• Traditional development approaches
  – fail to properly decouple computation from interaction
  – programming language specific
  – problematic for reconfiguration, extension, and scaling of software modules/systems
Evolving Software Components

• Software architecture building blocks
  – components, connectors, architectural configurations

• Claim
  – an existing software module can evolve in controlled manner via **subtyping**
Heterogeneous Subtyping

• Influenced by Object Oriented programming languages (OOPLs)
  – architectural component is similar to an OO class
• multiple, heterogeneous subtyping mechanism (PLs support a single mech.)
• reflect experience with components in the context of C2 architectural style
Heterogeneous Subtyping Approach

- Framework for defining subtyping relationships
- Based on a flexible type system
- Establishing *type conformance* of interoperating components
Heterogeneous Subtyping Approach (2)

• Each component specification treated as a type and its evolution is supported by subtyping
  – subtyping - evolution of a given type to satisfy new requirements
  – type checking - determines whether instances of one type can be replaced by instances of another type
Space of Type Systems

- Component subtyping relationships as regions in a space of type systems
Space of Type Systems - Labels

- $U =$ entire space of type systems
- Regions contain systems that demand two conforming types share
  - Int = Interface
  - Beh = Behavior
  - Imp = Implementation of all supertype methods
  - Nam = method names
Subtyping Relationships

• Subtyping relationships may be established by recognizing similarity of type space to set theory (hence, use of set operations)
• Several subtyping relationships are possible
• Can create new components by preserving some of the aspects of existing components
Behavioral Conformance
(Int and Beh)

• Demands that both interface and behavior of a type be preserved

\[ \text{Nam} \cap \text{Beh} \subset \text{Int} \cap \text{Imp} \]
Interface Conformance (Int)

- Demands that interfaces conform - without affecting dependent components
Strictly Monotone Subclassing (Int and Imp)

- Demands that both **interface** and **implementation** of a type be preserved
Implementation Conformance with Different Interfaces (Imp and not Int)

- Useful for using component in an alternate domain (e.g. domain translators in C2)
Type System - Formal Definition

- Formal specification definitions in Z
  - a language for modeling mathematical objects based on first order logic and set theory
    - logical connectives ($\land, \lor, \Rightarrow$, etc.)
    - set operations ($\in, \cup, \cap, \subseteq$)
Type System - Formal Definition

Component

- A component specification is an architectural type that may be instantiated multiple times in the configuration
- Components are distinguished from data they exchange
- Components are never passed from one component to another
Type System - Formal Definition Component (2)

- Name
- Set of internal state variables
- set of interface elements
- associated behavior
- implementation (?)
Type System - Formal Definition

Interface Element

- Name
- Direction (provided or required)
- set of parameters
  - Name
  - Type
- result (?)
Type System - Formal Definition

Behavior

• Invariant
  – specify properties that must be true of all component states

• Set of Operations
  – preconditions, postconditions, result (?)
  – set of variables
  – provided or required
Conclusive Remarks

• Pros
  – Help prove program correctness and completeness
  – Help resolve problems in software development
    (communication, coordination, ambiguity, etc.)
  – Easier to reconfigure and reuse software components

• Con
  – development of accompanying tools and software
  – diverse background and different viewpoints need different representations (e.g. client, managers, S/W eng.)
  – difficult to develop and understand for people having no background in formal methods and mathematics
  – ability to scale up to large real-world applications?

- A specification language provides facilities for explaining a program
  - Functional requirements for the program, i.e. mathematical description of what the program is required to do
  - Properties of its components as well as interactions between those components
  - Background knowledge
    - Description of the domain
    - Fundamental in
      - constructing programs
      - checking correctness of programs

• Design requirements of a specification language
  – Must be formalized
  – 2 Approaches:
    • *The fresh start* - does not have to accommodate the quirks of any given programming language design
    • *The evolutionary approach* - more likely to produce something that is used
    • Anna, an extension of Ada to support explanation

• Ada includes many useful constructs for specification
  – Subtypes, derived types, packages, generic units, constraints, exceptions, context specifications
    • increase program and compilation efficiency
    • reduce programming errors through readability
    • provide error checking at compile and run times
    • express programming decisions explicitly

• Ada is deficient in explaining programs and designs prior to implementation…

- Anna - a language extension of Ada providing added facilities for formal specification
  - 3 Categories of extensions
    - Generalization of explanatory constructs already in Ada
    - Addition of obvious new declarative constructs dealing with exceptions, context clauses, …
    - New specification constructs, e.g. AXIOMATIC SPECIFICATION

• Anna design considerations
  – Easy for Ada programmer to learn and use
  – Freedom to specify as much or as little as one desires
  – Extend Ada minimally, allowing for future incorporation of specification concepts
  – Encourage development of applications of formal specifications

• Resulting features include
  – Definition of a transformation of specifications into run-time checks
  – Axiomatic semantics
    • allows mathematical proof of consistency between Ada code and its formal Anna specification

- An Anna program is an Ada program with formal comments
  - Syntax is defined by extensions of Ada syntactic categories and new Anna syntactic categories
    - Ada point of view: only comments
    - Anna point of view: must obey syntax and semantics

- 2 Types of formal comments
  - Virtual Ada text - - :
  - Annotations - - |

- Virtual Ada text uses
  - Definition of programming concepts
  - Computation of values not computed by the actual Ada program

• Annotations
  – built-up from Boolean-valued expressions and reserved words
  – Types of annotations
    • Objects
    • Types or subtypes
    • Statements
    • Subprograms
    • Propagation of exceptions
    • Context annotations
    • AXIOMATIC ANNOTATIONS
    • …

- Quantified expressions
  - For all and exist (and negations)
  - Annotation is as rich as first-order logic
    - Concise and readable annotations
    - Establishing consistency between Ada code and Anna notations requires mathematical proof
      - Alternative run-time check is practical if domains of quantifiers are small

- Meaning of quantified Boolean expressions:

  \[
  \text{for all } X : T \Rightarrow P(X)
  \]
  means “for all values \(X\) of (sub)type \(T\), if \(P(X)\) is defined, then \(P(X)\) is true”

  \[
  \text{exist } X : T \Rightarrow P(X)
  \]
  means “there exists a value \(X\) of (sub)type \(T\) such that \(P(X)\) is defined and true”

- Axiomatic annotations
  - Constraints on operations of a package
  - Visible promises that may be assumed wherever the package specification is visible

- Example 1

```plaintext
package COUNTERS is

  type COUNTER is limited private;

  function VALUE (C : COUNTER) return NATURAL

  axiom for all C, D : COUNTER; N : NATURAL =>
  VALUE (C) = N and VALUE (D) = N => C = D;

end COUNTERS;
```

- The axiom requires the mapping VALUE to be one to one.

Ron Apcar
09/28/00

- Example 2
  - In Anna, the attribute ‘OUT is a record of all the output values produced by the subprogram.
  - \textbf{axiom for all} $E_0, E_1 : \text{ELEMENT}; Q_0 : \text{QUEUE} \Rightarrow$
    - REMOVE’OUT($E_0, \text{INSERT’OUT}(E_1,Q_0).Q).Q =$
    - INSERT’OUT($E_1, \text{REMOVE’OUT}(E_0,Q_0).Q).Q,$
    - LENGTH(INSERT’OUT($E_0,Q_0).Q) =
    - LENGTH($Q_0) + 1,$
    - LENGTH(REMOVE’OUT($E_0,Q_0).Q) =
    - LENGTH($Q_0) - 1,$
    - TOP(INSERT’OUT($E_0,Q_0).Q) = \text{TOP}($Q_0),
    - IS_MEMBER($E_0, \text{INSERT’OUT}(E_0,Q_0).Q),$
    - IS_MEMBER(\text{TOP}($Q), Q);

• Axiomatic semantics of Anna
  – Defines the atomic proof steps in mathematical proofs of consistency between Ada code and Anna notations
  – Basis for constructing program verifiers for Anna