Semantic Interconnection Models & Inscape

Dewayne E Perry
ENS 623A
Office Hours: T/Th 11:00-12:00
perry @ ece.utexas.edu
www.ece.utexas.edu/~perry/education/382v-s06/

Introduction

- Slow progress in management of software evolution
  - Increased use of tools, environments, automation
  - But, same underlying models
- One of the primary approaches used: the Interconnection Model
  - IM = ({ Objects } , { Relations })
- Currently used models
  - Unit interconnection model
  - Syntactic interconnection model
- Will introduce a new model
  - Semantic interconnection model

Unit Interconnection Model

- Basic Model
  - Unit IM = ({ units } , { "depends on" })
  - Units are typically files or modules
  - Basic relationship is one of dependency
- Utility
  - Supports and encourages modularity
  - Captures notion of encapsulation
  - Captures notion of localization

Example Uses of Unit IM

- Compilation contexts
  - C IM = ({ files } , { "includes" })
- Recompilation strategies
  - Ada IM =({ compilation-units }, { "with","changed more recently than" })
- Change notification
  - CN IM = ({ changed-units } , { "is used by" })
- System modeling
  - SM IM = ({ systems, files }, { "is composed with" })

Unit IM Evaluation

- Useful in a wide variety of contexts
- But, granularity is too large — units are generally composite objects
- Often need only part of the context supplied by this grain of object, e.g.,
  - May recompile too much
  - May broadcast change notification more than is necessary
- In general, want to compose systems out of smaller pieces than supported by the unit model

Thus, we need a finer grain of interconnection for more effective management of evolution in software systems

Syntactic Interconnection Model

- Basic Model

  Syntactic IM = (
  { functions, procedures, types, variables, ... } ,
  {"is used at", "is set at", "calls", "is called by",...}
)

  - objects are the syntactic elements in a programming language
  - relations reflect the basic uses of these objects
- Utility
  - localizes the interconnections used to those used in writing software
  - captures basic objects of evolution

Example Uses of Syntactic IM

- Change management

  Cross Reference IM = (
  { functions, procedures, types, variables, ...,
  locations} ,
  { "is defined at", "is set at", "is used at" }
)
- Static analysis - basic model to determine
  - semantic analysis
  - unreachable code, unset variables, etc.

Example Uses of Syntactic IM

- Smart recompilation

  Smart Recompilation IM = (
  { functions, procedures, types, variables, ... } ,
  {"is used at", ..., "is changed to", "is deleted from",
  "is added to"}
)
- System modeling

  System Modeling IM =(
  {systems, system-components, functions, procedures,
  types, variables, ...} ,
  {"is used at", "is set at", "calls", ..., "is composed with"}
)

Syntactic IM Evaluation

- Advantages over unit interconnection model:
  - Have a finer degree of interconnections and a richer set of relations
  - Have explicit interconnections between objects of program construction
- But, the following shortcomings:
  - Have no notion why these interconnections exist
  - No indication how the objects were intended to be used
  - No indication why the objects were in fact used

Thus we need to incorporate semantic information into our model in order to increase the effectiveness of our tools in managing the evolution of software systems

Semantic Interconnection Model

- Motivation
  - Need a way to express how objects are meant to be used and to capture how, in fact, they are used
  - Algebraic specifications are particularly apt for expressing relations between objects and how they are meant to be used
  - Input/output predicates express this aspect perhaps less elegantly, but are more suggestive about how objects are used
- Predicates and interconnections
  - Predicates provide the basic vocabulary with which to describe the behavior required and produced by system components
  - Metaphor for semantic interconnections: a component with its pre-conditions and post-conditions as a hardware chip with its input and output pins.

Semantic Interconnection Model

- Basic Model

```
Semantic IM = (
   {functions, procedures, types, variables, ..., predicates} ,
   {"is used at", "is set at", "calls", "is called by", ...,
    "satisfies"}
   )
```

- Example uses in The Inscape Environment
  - program construction
  - program evolution
  - version control

Inscape Semantic IM

```
Inscape IM = (
   {..., preconditions, obligations, postconditions, exceptions} ,
   {..., "satisfied by", "satisfies", "propagated", "precludes",
    "handles", ...}
   )
```

- Pre-conditions/obligations are satisfied by post-conditions or are propagated to the interface
- Post-conditions satisfy preconditions/obligations and are propagated to the interface
- Failure of some preconditions can be handled by exception handlers
- Some exceptions can be precluded by the satisfaction of their related preconditions

Semantic IM Evaluation

- Construction
  - Enforce the consistent use of the Instress specifications
  - Record the relationships between predicates
  - Enforce the semantics of program construction with respect to the specified interfaces
- Evolution
  - To determine the implications and extent that changes to interfaces have on implementations — for example,
    - whether there is any effect at all
    - whether code is no longer needed
    - whether new code is required
  - To determine the implications and extent that changes to implementations have on their interfaces
  - To provide facilities for simulating and propagating changes
  - To guarantee the consistency and completeness of changes

Summary

- The semantic interconnection model incorporates the advantages of the unit and syntactic interconnection models and provides extremely useful extensions
  - Formalize the semantics of program construction
  - Provide a deeper understanding of the semantics of change
  - Define intuitive notions of version equivalence and compatibility

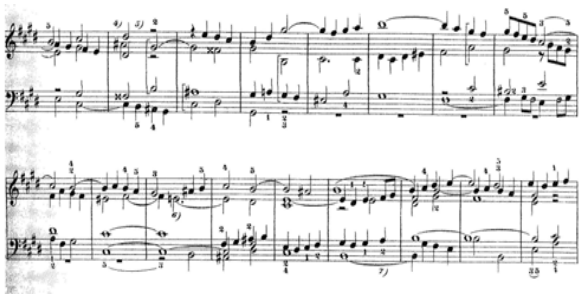
With this model, we provide tools that are knowledgeable about the process of system construction and evolution and that work in symbiosis with system builders to construct and evolve large systems

The Inscape Environment

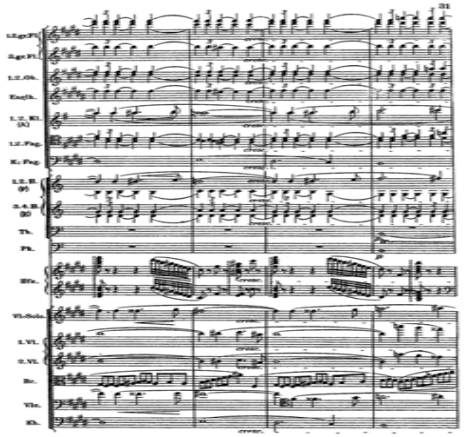
Background

- Complexity
  - Basic issues
    - No two parts alike - ie, all parts distinct
    - Scale up by addition, not replication
    - Very large number of states – hard to conceive, understand
  - 2 kinds of complexity
    - Intricacy
      - ✓Particularly true of algorithms
      - ✓Like a Bach 4 voice fugue
        - Horizontal and vertical relationships
        - Hard to change one note without severe repercussions
    - Wealth of detail
      - ✓Nothing very deep, just masses of details
      - ✓Like a Strauss tone poem, or Mahler symphony
        - Massive number of notes on a page – provide texture
        - Missing one would hardly be noticed
      - ✓Makes very hard to comprehend the entire system (eg, 10M lines)

Complexity: Intricacy (Bach)



Complexity: Wealth of Detail (Strauss)



Background

- → Model of software development environments
  - SDE model = { policies, mechanisms, structures }
- → 4 classes of SDEs
  - Individual:
    - Primary issue: construction
    - Mechanisms dominate
  - Family:
    - Primary issue: coordination
    - Structures dominate
  - City:
    - Primary issue: cooperation
    - Policies dominate
  - State:
    - Primary issue: commonality
    - Higher-order policies dominate

Inscape Overview

- → A City Model SDE
  - Constructive use of module interface specifications
  - Environment supported crowd control
- → Concerned about:
  - Complexity: intricacy, wealth, and invisibility of detail
  - Evolution: not just a matter of "getting it right the first time"
  - Scale: lengthy, large projects with large groups of people

Inscape Approach

- Models of software interconnections
- Models of software development environments
- Formalize the software development process (specifically system construction and evolution)
- Practical use of specification and verification technology: the constructive use of specifications
- Incorporate intelligence/understanding into the environment:
  - of specifications
  - of the software development process
  - of the implementation language

Research Strategy

- Systematically work out the implications of using formal interface specifications.
- Dance around the "tarpit" of verification
  - use "shallow" consistency checking (ie, tend toward pattern matching, simple deductions)
  - automate as much as possible
  - interact where cannot automate
- Use incremental techniques to distribute the cost of analysis
  - fine grain: statement
  - large grain: operation

Structure of the Inscape Project

- Instress Interface Specification Language
- Inform System Construction
- Intertwine Interface Propagation
- Inquire Project/System Browser/Search Mechanism
- Infuse System Evolution
- Integrate Integration/Regression Testing
- Invariant Version Management

Module Specifications — Basis

- Basis: Hoare Input/Output Predicates
- Extensions:
  - Obligations — "entailed" in addition to "known"
  - Multiple Results — exceptional in addition to normal results
  - Three flavors of Preconditions (because of relationship with exceptional results):
    - assumed - assumed to be true; must satisfy or propagate
    - validated internally validated preconditions; must either satisfy or handle the exception
    - dependent truth not knowable until an attempt (without special knowledge, no way to satisfy); must handle exception

Module Specifications - Details

- Vocabulary of the abstraction
- Data objects and properties
- Relationships among data objects
- Operations - effects and side effects
- Relationships among operations
- Relationships among operations and data
- Exceptions - effects and side effects
- Exception recovery

Module Specifications

- Instress — the module interface specification language
  - specification logic (SL)
  - C syntax for declarations
  - SL annotations for data object properties and operation interface behavior
  - pragmatic information
- Analysis - Instress analysis ensures that
  - Predicate definitions are consistent Precondition, Postcondition, Obligation, and Property lists are consistent
  - Validated and Dependent Preconditions are represented as Failed Conditions in at least one exception
  - <in> parameters have Preconditions
  - <out> parameters have Postconditions

Construction — Semantic Basis

- Semantic Dependencies
  - Origins
    - operation instantiation
    - user-specified assertions
    - object properties
  - Interconnections, Dependencies
    - postconditions satisfy preconditions
    - postconditions satisy obligations
- Interface Propagation
  - Incremental Basis
    - unsatisfied preconditions/obligations
    - accumulated postconditions
  - Function of Components
    - sequence sequents
    - operation local declarations and implementation sequence
    - selection boolean expression, then and else sequences
    - Iteration boolean expression and loop bodysequence

Construction — Control Flow Basis

- Formalization of Exception Handling:
  - Precluded — associated failure condition has been satisfied - no need to handle the exception.
  - pruned — conscious refusal - associated preconditions become assumed
  - reported — exception propagated, possibly with repair, to the interface.
  - recovered — exception handled by retrying the operation, possibly with repair
  - repaired — results of exception fixed & merged with successful results the successful case
  - ignored — results of the exception are satisfactory & merged with successful results
  - coalesced — two or more exceptions merged and propagated
  - introduced — arbitrary result considered exceptional and propagated, possibly with repair

Construction — Analysis

- Basic rule: preconditions and obligations must be satisfied or propagated to the interface
- Logical barriers: precondition ceilings, obligation floors (cannot be propagated, hence, must be satisfied)
- Errors in looping structures: postconditions from loop form a logical barrier to preconditions at point of re-entry
- Completeness of implementation: empty precondition ceilings and obligation floors, and no looping errors
- Correctness: propagated interface "matches" specified interface

Evolution

- Semantic Interconnection details
  - precondition dependence and propagation
  - obligation dependence and propagation
  - postcondition satisfaction and propagation
- Exception handling
  - exception preclusion
  - methods of exception handling
- Kinds of Changes:
  - Predicate definitions - worst case
  - Precondition, obligation, postcondition, property additions/deletions
  - Exception failed condition addition/deletion
  - Exception addition/deletion
- Effects on Implementation
  - Dependency re-analysis
    - no effect
    - code no longer needed
    - new code needed
  - Exception analysis
    - effects on preclusion, removal, handling, new handling decisions

Evolution

- Affects implementation and interface
  - Semantic Dependencies
    - add/remove satisfaction/dependence
    - add/remove logical barriers
    - alter propagated predicates (ie, interface)
- Exception Handling
  - propagated → internal
  - internal → propagated
  - * → pruned - may affect propagated interface
  - * → ignored - may affect propagated postconditions, obligations
  - * → precluded - may affect propagated postconditions, obligations

Evolution

- Crowd Control
  - Addresses Issues:
    - How to suport large numbers of programmers
    - How to manage the change process
  - Choreographed interactions
    - policies: enforced cooperation
    - mechanisms: dependence-order clustering, change propagation
    - structures: hierarchical experimental databases
  - Unconstrained interactions
    - policies: voluntary cooperation
    - mechanisms: user-selected population, change simulation
    - structures: workspaces

Use and Reuse

- Predicate-based assistence
  - Find objects to satisfy preconditions & obligations, produce desired behavior
  - Provide associated cost of objects
  - Use unit, syntactic, and semantic interconnections for browsing
- Predicate-based plug-compatibility
  - Version equivalence
  - Version compatibility - dependency & function preservation
  - Implementation compatibility – effect on propagated interface

Validation

- Incremental static analysis
  - weaken the specification logic to make it more tractable
  - weaken consistency checking to make it tractable
  - propagation - modal propositional calculus
- Incremental dynamic analysis
  - marries module test harness with crowd-control structure
  - semi-automatic construction of integration test harness
  - automatic selection of regression tests
  - interactive extension of regression tests

Research Contributions

- Obligation and Multiple Result Specifications
- Semantic Interconnections and Propagated Interfaces
- Enforcing consistent use of Interface Specifications
- Change implication in Interfaces and Implementations
- Enforced and Voluntary Cooperation mechanisms and structures
- Integration Test Management
- Predicate-based Assistance
- Formalization of Version Management concepts