Behavior Extraction
related work

Tomáš Poch
DISTRIBUTED SYSTEMS RESEARCH GROUP
http://dsrg.mff.cuni.cz
CHARLES UNIVERSITY PRAGUE
Faculty of Mathematics and Physics
Outline

• Introduction, Context

• Approaches in general
• 3 papers

• Conclusion
Goal, Motivation

• Extract the high-level specification from the existing implementation

• Use the extracted specification for various sw engineering purposes
  ▪ Documentation
  ▪ Performance analysis
  ▪ Formaly ensure certain properties (error detection)

• Manual creation of specifications is
  ▪ Error prone
    • Does not guarantee any relation to implementation
  ▪ Costly
Expressiveness Gap

• Expressive power of target formalism is often small

• Extraction must simplify the behavior
  ▪ Enaugh – so that complexity vanishes
  ▪ Not too much – so that important properties are kept
  ▪ Over-specification – capture more behavior that the implementation actually exhibits

• Since the components encapsulate the complex behavior, a reasonable result should exist
Static component analysis

- Allows to extract usage and effects
  - ! & ? In BP

- Steps
  - Create a component abstraction
    - Separate model for each method, shared variables \(\rightarrow\) effects specification
  - Identify abstract error states
    - Predicate over abstract states
  - Derive allowed usage

Lie et al. *A simple method for extracting models from protocol code*,

Whaley et al. *Automatic extraction of object-oriented component interfaces*,
Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, July 22-24, 2002, Roma, Italy

Alur et al. *Synthesis of interface specifications for Java classes*,
ACM SIGPLAN Notices, v.40 n.1, p.98-109, January 2005
Static client analysis

- Assumption: client source is available and exercises correct behavior
- Allows to extract usage specification
  - BP question marks

Shoham et al. Static specification mining using automata-based abstractions
Proceedings of the 2007 international symposium on Software testing and analysis, July 09-12, 2007, London, United Kingdom
Dynamic analysis

• Assumption
  ▪ Both, client and component implementations are available
  ▪ Client does not have to be necessarily correct

• Two phases
  ▪ Run an application containing the component
    • Use monitoring to collect traces of method calls on a component boundaries
  ▪ Derive a specification from the set of traces
    • Include identification of common patterns

• Cons
  ▪ The result specification allows just the behavior exercised by the application
  ▪ Need to compile, deploy and run the application

Whaley et al. *Automatic extraction of object-oriented component interfaces*,
Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, July 22-24, 2002, Roma, Italy

Ammons et al. *Mining specifications*,

Lorenzoli et al. *Inferring state-based behavior models*,
Proceedings of the 2006 international workshop on Dynamic systems analysis, May 23-23, 2006, Shanghai, China
Handy keywords

- Static analysis
  - Parse tree transformations
  - Abstraction
    - Predicate
    - Heap
    - History
  - Points-to analysis

- Model checking
- Monitoring
- Game theory

- Orthoschist (Ortobřidlice)
A Simple Method for Extracting Models from Protocol Code

David Lie Andy Chou Dawson Engler David L. Dill
Computer Systems Laboratory
Stanford University
Stanford CA 94305
E-mail: fdavidlie,acc,engler,dillg@stanford.edu
• Goal
  - Extract Murφ high level specification of an C code
• Means
  - Slicing
  - Rewriting rules on AST
• Tools
  - xg++
    - Compiler, allows to write domain specific extensions (extraction, translation)
  - Murφ
    - Spec. language (Pascal-like syntax)
    - Explicit state space model checker
• Application
  - FLASH cache coherence protocol
    - large-scale distributed multiprocessor architecture
Application

User steps

- Define the program state
  - List of important variables and functions
- Define the transformations (rewriting rules)
  - User may specify special transformations
    - Abstraction, Checks
- Create a model of HW, correctness properties and initial state

Figure 1. Flow chart of model extraction and verification
Conclusion

• The extraction requires still a lot of effort
  
  ▪ State identification, abstraction, and property specification is done by hand.
    ▪ If the abstraction and slicing does not eliminate all complex stuff, the process fails

• Amount of manual labor required is reduced
• The effort can be reused, when implementation changes
• Properties may be checked independently
  
  ▪ Derived model is different depending on slicing

• Application – 8 bugs found
Automatic Extraction of Object-Oriented Component Interfaces
John Whaley Michael C. Martin Monica S. Lam
Computer Systems Laboratory
Stanford University
{jwhaley, mcmartin, lam}@stanford.edu
Goal

- Model the interface of a class (? in BP) by multiple FSMs
- Derive interface models from code
  - Employ static analysis
  - Employ monitoring
- Use the result for runtime-checking, documentation
Interface model

- Multiple FSMs
  - Each FSM models a subset of methods
    - Implementing certain interface or
    - accessing certain fields
  - FSM
    - state – a method modifying the state of the component
    - transition – a allowable pair of consecutive methods
    - set of states for each side-effect-free method
      - in those states the method can be invoked
Extraction algorithm

- Assumption
  - Class under analysis is correct \( \rightarrow \) defensive implementation

- Steps
  1. For each method \( m \) identify predicates that guards throwing of exceptions
  2. Find methods \( n \) that sets the fields to values that may cause an exception in \( m \)
     - Transition from \( m \) to \( n \) is illegal

- Undecidable in general \( \rightarrow \) restrictions
  - Step 1 – guards in form
    - field = constant;
  - Step 2 – transition is illegal, iff
    - \( n \) must assign the value

- Illegal transitions imply exceptions
  - Good for error detection
  - Does not ensure correctness

Tomáš Poch

Java.util.AbstractList.ListItr
Dynamic extraction

- The model is fairly simple
  - Important are just consecutive calls
  - No need to employ FSM learning algorithm

- Instrumentation of bytecode
  - Method calls on `this` are ignored

- Runtime checking
Conclusion

• Although the static extraction is very restricted it works in many cases
  ▪ Java classpath
    • Vector and LinkedList
    • I/O and socket classes
    • Timer
    • SimpleTimeZone
    • ThreadGroup
    • Signature

• If the static approach fails, dynamic may be used
Synthesis of Interface Specifications for Java Classes
Rajeev Alur Pavol Černý
P. Madhusudan Wonhong Nam
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104
{alur, cernyp, madhusudan, wnam}@cis.upenn.edu
Goal

- Automatically generate temporal specification of Java class usage from the class implementation

- Uses predicate abstraction, model checking, and FSM learning (L*)

- The tool is called JIST
• **Java2Jimple**
  - Jimple – 3-address representation of bytecode (Soot framework internal representation)
• **Jimple to Boolean Jimple**
  - Employ predicate abstraction to simplify the implementation. Predicates are provided by user
• **Boolean Jimple to NuSMV**
• **Interface synthesis**
  - L* algorithm is learning the automaton from answers NuSMV
  - Error predicate is provided by user
Predicate abstraction

- Predicates are specified by user
  - Just in form \( \text{var} = \text{constant} \) (again)
- Abstraction is done automatically on Jimple
- Result is a boolean program - 3 state logic
  - Variables in form \( b(x,k) \in \{\text{true, false, *}\} \)
    - \( x == k \) evaluates to true/false/unknown

- Notions needed by abstraction for each statement
  - \( \text{WP}(st, P) = Q \) (weakest precondition)
    - \( P, Q \) is predicate, \( st \) is statement, \( \text{WP} \) is function
    - if \( Q \) holds, after execution if \( st \) will hold, \( Q \) is minimal
  - \( \text{Implies}(Pr)(P) = Q \)
    - \( Pr \) is the set of predicates provided by the user, \( P \) is predicate, \( Q \subseteq Pr \)
    - \( Q => P \), \( Q \) is maximal

- Transformation
  - \( b(p) \rightarrow \text{Implies}(Pr)(\text{WP}(st,p))? \text{true : Implies}(Pr)(\text{WP}(st, not p))? \text{false : *} \)
Predicate abstraction example

- $Pr = \{(y, l_1), \ldots, (y, l_n)\}$ – set of predicates involving the $y$ variable
- Statement $x = y + 1$ is replaced by a sequence of statements.
  - For each predicate in form $(x, k)$
    - if $(y, k-1) \in Pr$
      - $b(x, k) = b(y, k-1)$
    - otherwise
      - if $(b(y, l_1) == true \lor \ldots \lor b(y, l_n) == true)$
        - $b(x, k) = false;$
      - else $b(x, k) = *;$

- Method calls are either inlined or abstracted in a coarse way if there is no implementation (JML might help)
  - $x = f() \rightarrow b(x, k_1) = * \; ; \; \ldots \; ; b(x, k_n) = *$
Interface synthesizer

• User provides the error predicate E
• M – set of methods provided by the class
• Boolean program BP defines a language $L(BP)$ over M
  - $m \in L(BP)$ iff the class executing the method sequence $m$ does not reach an error state (a state satisfying E)

• Boolean program can be translated into NuSMV input language

• Interface synthesizer is using L* algorithm
  - FSM learning algorithm
  - Queries the model checker
    - “is w member of L(BP)?” (membership query)
    - “is the language generated my current result $R$ equal to the $L(BP)$?” (equivalence query)
  - Counter examples are used to refine the result
Model checker queries

• Membership is transformed into subset query
  ▪ $m \in L(BP)$ iff $\{m\} \subseteq L(BP)$
• $L(R) \subseteq L(BP)$ is done by CTL checking
  ▪ State spaces of $R$ and $BP$ are combined
  ▪ CTL formula $AG(\neg E)$ is examined

• For equivalence query we need also $L(R) \supseteq L(BP)$ (superset query)
  ▪ For each sequence of methods $m$
    • $m \notin L(R) \Rightarrow m \notin L(BP)$
    • $m \notin L(R) \Rightarrow$ there is a run of the boolean program that does not stay within safe states
• Author claims, that the task is NP-complete
  ▪ Simpler question, which implies the previous one
    • $m \notin L(R) \Rightarrow$ all runs of the boolean program does not stay within safe states ($\psi$)
  ▪ State spaces are combined in other way
    • captures legal method sequences followed by one illegal
    • $CTL : AG((\text{legal} = 0 \&\& R \text{ is on turn}) \rightarrow EF (E))$
    • Once a state is reached when the illegal method was called, then there is a path to an error states
  → In some cases, the generated interface is not the most general
Results

- Java classpath
  - ListItr
  - ServerTableEntry
  - Signature
  - PipedOutputStream

Figure 5: List Iterator (1 predicate)
Conclusion

- Approaches combine algorithms from different areas
- None of them solves the problem utterly
  - Part of the problem is solved in special cases

- Tools are getting better
  - Theory behind is getting complex

- We can try to use them in Q-Impress
  - Class → Component step must be done
  - Technical issues (metamodel)
  - We can try to improve them