Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer

DAVOR GABERŠEK
Overview

• Semantic framework
• Techniques for scenario control
• Test generation
• Test execution
• Summary
• Conclusion
Semantic framework

• States
  • Described by function symbols whose interpretation may change – state variables or dynamic functions. All state variables are either nullary or unary.

• Model Automata:
  • As an extension of interface automata
  • Includes the concept of accepting states familiar in formal languages
  • The paper considers only deterministic model automata (although reactive systems are inherently nondeterministic)

• Model programs
  • Can be written in AsmL, Spec#, C# or even Visual Basic
  • Defines the possible behavior of the program (update rules of the abstract state machine)
    • Defines actions – observable or not with [Action] or [Action(Kind=ActionAttributeKind.Observable)]
Semantic Framework

• Model programs
  • Behavior definition
    • Prerequisites (keyword requires)
    • Special high-level datatypes (i.e Map or Seq)
    • Denotations of instance types (enumof(T))

• Example:
Spec# example:

```csharp
Class Client {
    bool entered;
    Map<Client, Seq<string>> unreceivedMessages;

    [Action] Client() {
        this.unreceivedMessages = Map;
        foreach (Client c in enumof(Client), c != this){
            c.unreceivedMessages[this] = Seq{};
            this.unreceivedMessages[c] = Seq{};
        }
    }

    entered = false;
}

[Action] void Enter()
    requires !entered; {
        entered = true;
    }

[Action(Kind=ActionAttributeKind.Observable)]
void Receive(Client sender, string message)
    requires sender != this &&
    unreceivedMessages[sender].Length > 0 &&
    unreceivedMessages[sender].head == message; {
        unreceivedMessages[sender] = unreceivedMessages[sender].tail;
    }
}
```

...
Model automata example

• Two clients – c0 and c1

• Each client sends a message „hi from ...“
Semantic framework

- State Exploration
  - Actions not considered as abstract labels but have an internal structure
  - Invocation of an action with a parameter in a state $s$ is formalized using the Abstract State Machine theory

- Example:
  - We invoke the Client constructor in state $s_0$ – the precondition is true (no precondition).
  - This invocation produces an update that adds a new object – i.e $c_0$ to the dynamic universe Client and $s_1$ is the resulting state:
    $\delta(s_0, \text{Client}()\!/c_0) = s_1$
  - From state $s_1$ we can continue by invoking $c_0$.Enter()
  - All the prerequisites are met ( $c_0$ is a type of Client and $c_0$.entered equals false), therefore the $c_0$.Enter() is enabled in $s_1$, making a new state – $s_2$
    $\delta(s_1, c_0$.Enter()) = s_2$
Semantic framework

• Controllable and observable actions
  • In reactive systems there are no input and output methods we can access, therefore we determine that an action can be controlled (an input) or only observed (output)

• Accepting states
  • A state s is an accepting state if the accepting state condition is true in s
  • Is a closed Boolean state based expression (true or false)
  • Example: excluding the initial state of the client where no messages have been received:

    enumof(Client).Size>0 && Forall{ c in enumof(Client), s in c.unreceivedMessages.Keys; c.unreceiveMessages[s].Length ==0 }
Semantic framework

• State invariants
  • A closed Boolean expression that must hold at all states
  • If the model program violates a state invariant (the state invariant being false in some state of the model) then this model program is not valid.
  • It is a safety condition on the transition function or an axiom on the reachable space that must always hold

• Example:
  Forall{ c in enumof(Client); c notin c.unreceivedMessages.Keys } 

Meaning that no client should be considered as a recipient for his own messages.
Techniques for Scenario Control

- Techniques for selectively exploring transitions of the model program

  *Parameter selection* – limits exploration to a finite but representative set of parameters for the action methods
  - The Spec Explorer provides a user interface for parameter selection with four levels of control: **Default, Per Type, Per Parameter** and **Per Method**

- **Method restriction** – removes some transitions based on user-provided criteria
  - Limiting scenarios included in our transition system by strengthening the preconditions of an action.
  - I.e: Adding an auxiliary type representing the mode of the system
    - That way we can restrict the enabling of the actions Client, Enter and Send to those states where we want to see them
Techniques for Scenario Control

- **State filtering** – prunes away states that fail to satisfy a given state-based predicate
  - Spec Explorer allows users to specify the set of state filters. If $e(s)$ is true, the state $s$ is considered to be in the state filter $S_f$.

- **Directed search** – performs a finite-state walk of transitions with respect of user-provided priorities. State transitions that are no visited are pruned away.
  - Weights are given in Spec Explorer as state-based expression that return non-negative integers.
Techniques for Scenario Control

- **State grouping** – selects representative examples of states from user-provided equivalence classes
  - Spec Explorer visualizes a state grouping $G$ of model automation $M$ as a graph. The nodes of the graph are elements of $S/G$.
  - Example: 3 clients, each sends two messages: "hi" and "bye".
Test generation

• Model based test generation and test execution are two closely related processes

• The traditional view on test generation test are generated in advance from a given specification or a model to provide some kind of coverage of the state space, to reach a state satisfying a particular property or to generate random walks in the state space. This is also called *offline testing*
  • test execution is a secondary process, using pre-generated tests and ran against an implementation.
Test generation

• In another extreme, both processes are intertwined into a single process where tests are generated on-the-fly as testing progresses. This is called on-the-fly testing or online testing.

• In Spec explorer tests are represented explicitly in the offline case as sets of action sequences called test segments. These are linked together to encode branching with respect to observable actions.

• Algorithms discussed in Optimal strategies for nondeterministic systems (2004) are used in Spec explorer for transition coverage and the purpose of reaching states. Spec Explorer uses the value iteration algorithm for negative Markov decision problems to generate tests that optimize the expected cost where the observable actions are given probabilities.
Test generation

Online test generation – in Spec explorer the model program is used for on-the-fly testing. Action weights are used to select controllable actions from among a set of actions for which parameters have been generated from specific generators.
Test execution

- The testing process assumes that the implementation under test is encapsulated in an observationally complete “wrapper”.
- This is needed to be able to guarantee termination of test execution.
- For observational completeness the spec explorer uses a timeout action, which is approximated by using a state based expression that determines the amount of time to wait for observable actions to occur.
- Spec Explorer provides a mechanism for the user to bind the action methods in the model to methods with matching signatures in the IUT and enabledness is validated generating input parameters so that preconditions are met for a method to be called and after the call the results given are compared to the expected results.
Test execution

- Conformance automaton – is a machine that takes a model M, a test T and an observationally complete implementation wrapper N and then executes each test in T against N. The automaton keeps track of the set of object bindings.
  - The verdict of this automaton can have the following values: Undecided, Inconclusive, Failed, TimedOut and Succeeded.
  - The implementation of Spec Explorer did not contain the Inconclusive verdict, saying that it is the tester responsibility to guarantee that the test is complete or to tolerate a failure verdict also for inconclusive tests.
Summary

• Model programs compactly encode large transition systems

• Exploration produces a model automation
  • The model program can be unwound into a model automation

• Traversal of automata underlies automatic test generation
  • Test cases generated by traversing the graph of the model automaton

• Conformance can be formally defined as alternating simulation
  • Differences between predicted and actual behavior are conformance failures

• Test execution implements the conformance relation
  • Test graphs are also used to automatically harness the implementation for conformance testing
Conclusion

• Spec Explorer covers a broad range of problems and solutions in the domain, including dynamic object creation, non-determinism and reactive behavior, model analysis, offline and online testing and automatic harnessing.

• It is used on a daily basis internally at Microsoft, however improvements are necessary.

• Improvement suggestions:
  • Scenario control – currently realized by parameter generators, state filters, method restriction, etc. Sometimes the scenario control can be more challenging than describing the functionality of the test oracle. Fragments of the scenario control are spread over various places in the model. An improvement would be to centralize all scenario control related information as one „aspect“ in a single document.
Conclusion

• Improvement suggestions:
  • Model composition – Integration testing of separate modules should be tested together, so users would like to be able to compose compound model from existing models.
  • Symbolic exploration – the Spec Explorer requires use of ground data in parameters provided for actions, which is a restriction and is sometimes required only by underlying technical constraints of the tool. The new generation of the tool will be having a symbolic approach for generalizing exploration.
  • Measuring coverage and testing success – a problem of model-based testing is developing adequate coverage and test sufficiency metrics. Coverage becomes difficult in internal non-determinism.
Conclusion

• Improvement suggestions:
  • Continuing after failures – a failure in model based testing means that testing cannot be continued, however when a bug is discovered it would be more practical to continue the testing even in presence of bugs, since the time from the bug discovery and bug fixing can be rather long.

• Verdict:
  • Spec Explorer has shown that model-based testing can be very useful and can be integrated into the software development process.
  • The main directions for further improvement are:
    • Compositional modeling
    • Improved online algorithms and
    • Symbolic execution.