Extended Interface Grammars for Automated Stub Generation

Graham Hughes    Tevfik Bultan
Department of Computer Science
University of California, Santa Barbara
Outline

• Motivation
• Interface Grammars
• Shape Types
• Interface Grammars + Shape Types
• Experiments
• Conclusions
Motivation

Cool Verification Technique

A Software System
Motivating Examples

• **Cool verification technique**: Action Language Verifier
  – An infinite state model checker for specifications with unbounded integers, boolean and enumerated variables

• **Application**: Check synchronization in Java programs

• Does not really work
  – ALV cannot handle Java semantics (objects, recursion etc.)
  – ALV would not scale to the state space of a typical Java program
module main()

    integer nr;
    boolean busy;
    restrict: nr>=0;
    initial: nr=0 and !busy;

module ReaderWriter()
    enumerated state {idle, reading, writing};
    initial: state=idle;

    r_enter: state=idle and !busy and nr'=nr+1 and state'=reading;
    r_exit:  state=reading and nr'=nr-1 and state'=idle;
    w_enter: state=idle and !busy and nr=0 busy' and state'=writing;
    w_exit:  state=writing and !busy' and state'=idle;

    ReaderWriter: r_enter | r_exit | w_enter | w_exit;
endmodule

main: ReaderWriter() | ReaderWriter() | ReaderWriter();

spec: invariant(busy => nr=0)
spec: invariant(busy => eventually(!busy))

endmodule
class ReadWriteLock {
    private Object lockObj;
    private int totalReadLocksGiven;
    private boolean writeLockIssued;
    private int threadsWaitingForWriteLock;

    public ReadWriteLock() {
        lockObj = new Object();
        writeLockIssued = false;
    }

    public void getReadLock() {
        synchronized (lockObj) {
            while ((writeLockIssued) || (threadsWaitingForWriteLock != 0)) {
                try {
                    lockObj.wait();
                } catch (InterruptedException e) {
                }
            }
            totalReadLocksGiven++;  
        }
    }

    public void getWriteLock() {
        synchronized (lockObj) {
            threadsWaitingForWriteLock++;
            while ((totalReadLocksGiven != 0) || (writeLockIssued)) {
                try {
                    lockObj.wait();
                } catch (InterruptedException e) {
                    //
                }
                threadsWaitingForWriteLock--;
                writeLockIssued = true;
            }
        }
    }
}
Motivating Examples

- **Cool Verification Technique**: Java Path Finder
  - An explicit state model checker (like Spin) for Java programs

- **Application**: Check assertions in Java programs

- Does not really work
  - JPF cannot handle native code
  - JPF does not scale to large Java programs
Verifiability Via Modularity

- **Modularity** is key to **scalability** of any verification or testing technique
  - Moreover, it can help **isolating the behavior** you wish to focus on, removing the parts that are beyond the scope of your verification technique

- Modularity is also a key concept for successful software design
  - The question is finding effective ways of exploiting the modularity in software during verification
Interfaces for Modularity

- How do we do *modular verification*?
  - Divide the software to a set of modules
  - Check each module in isolation

- How do we *isolate a module* during verification/testing?
  - Provide *stubs* representing other modules

- How do we get the stubs representing other modules?
  - Write *interfaces*
    - Interfaces specify the behavior of a module from the viewpoint of other modules
    - Generate stubs from the interfaces
Interface Grammars

- Interface Grammar
- Component
  - Program
- Interface Grammar
  - Interface Compiler
  - Component Stub
    - Program
- Model Checker
An Example

• An *interface grammar for transactions*
  – Specifies the appropriate ordering for method calls to a transaction manager
  – Method calls are the terminal symbols of the interface grammar

```
Start → Base
Base → begin Tail Base
     | ε
Tail → commit
     | rollback
```
An Example

- Consider the call sequence
  \texttt{begin rollback begin commit}
- Here is a derivation:

\begin{align*}
\text{Start} & \Rightarrow \text{Base} \Rightarrow \textbf{begin} \text{ Tail Base} \\
& \Rightarrow \textbf{begin} \textbf{ rollback} \text{ Base} \\
& \Rightarrow \textbf{begin} \textbf{ rollback} \textbf{ begin} \text{ Tail Base} \\
& \Rightarrow \textbf{begin} \textbf{ rollback} \textbf{ begin} \textbf{ commit} \text{ Base} \\
& \Rightarrow \textbf{begin} \textbf{ rollback} \textbf{ begin} \textbf{ commit} \\
\end{align*}

\begin{align*}
\text{Start} & \rightarrow \text{Base} \\
\text{Base} & \rightarrow \textbf{begin} \text{ Tail Base} \\
& | \epsilon \\
\text{Tail} & \rightarrow \textbf{commit} \\
& | \textbf{rollback}
\end{align*}
Another Example

- The earlier example we gave can also be specified as a FSM
- However, the following grammar which specifies *nested transactions* cannot be specified as a FSM

```
Start → Base
Base → begin Base Tail Base
|   ε
Tail → commit
| rollback
```
Yet Another Example

• Let’s add another method called **setrollbackonly** which forces all the pending transactions to finish with **rollback** instead of **commit**

• We achieve this by extending the interface grammars with **semantic predicates and semantic actions**

\[
\begin{align*}
\text{Start} & \rightarrow \quad \langle r:=\text{false}; \; l:=0 \rangle \; \text{Base} \\
\text{Base} & \rightarrow \quad \text{begin} \; \langle l:=l+1 \rangle \; \text{Base} \; \text{Tail} \\
& \quad \langle l:=l-1; \; \text{if } l=0 \; \text{then } r:=\text{false} \rangle \; \text{Base} \\
& \quad \mid \; \text{setrollbackonly} \; \langle r:=\text{true} \rangle \; \text{Base} \\
& \quad \mid \; \epsilon \\
\text{Tail} & \rightarrow \quad \langle r=\text{false} \rangle \; \text{commit} \\
& \quad \mid \; \text{rollback}
\end{align*}
\]
Our Interface Grammar Language

rule base {
  choose {
    case ?begin: {
      «l++;»
      return begin;
      apply base;
      apply tail;
      «l--; if (l==0) r=false;»
      apply base;
      case ?setRollbackOnly: 
      «r=true;»
      return setRollbackOnly;
      apply base;
      ...
    }
  }
  ...
}
Verification with Interface Grammars

- Interface Grammar
- Interface Compiler
- Component Stub
  - Parser stack
  - Parse table
  - Top-down parser
    - Semantic predicates and semantic actions
- Program
- Model Checker
  - Method invocation (lookahead)
Checking Arguments

• A crucial part of the interface specification is specifying the allowable values for the method arguments and generating allowable return values.

• In what I discussed so far all these are done in the semantic actions and semantic predicates.

• The question is can we specify the constraints about the arguments and return values using the grammar rules.
  – *Recursive data structures* are especially good candidates for this!
Shape Types

• Shape types [Fradet, Metayer, POPL 97] provide a formalism for specifying recursive data structures

• It is a specification formalism based on graph grammars

• Shape types can be used to specify the connections among the heap allocated objects

• Objects become the parameters of the nonterminals and the constraints on the connections among the objects are specified on the right-hand-sides of the grammar rules (similar to semantic predicates)
Shape Type for Doubly Linked List

\[
\begin{align*}
Doubly & \rightarrow \ p \ x, \ \text{prev} \ x \ \text{null}, \ L \ x \\
L \ x & \rightarrow \ next \ y, \ \text{prev} \ y \ x, \ L \ y \\
L \ x & \rightarrow \ next \ x \ \text{null}
\end{align*}
\]

\[
\begin{align*}
Doubly & \Rightarrow \ p \ 1, \ \text{prev} \ 1 \ \text{null}, \ L \ 1 \\
& \Rightarrow \ next \ 1 \ 2, \ \text{prev} \ 2 \ 1, \ L \ 2 \\
& \Rightarrow \ next \ 2 \ 3, \ \text{prev} \ 3 \ 2, \ L \ 3 \\
& \Rightarrow \ next \ 3 \ 4, \ \text{prev} \ 4 \ 3, \ L \ 4 \\
& \Rightarrow \ next \ 4 \ \text{null}
\end{align*}
\]
Shape Type for Binary Tree

\[
\begin{align*}
B\text{intree} & \rightarrow p \, x, \, B \, x \\
B \, x & \rightarrow \text{left} \, x \, y, \, \text{right} \, x \, z, \, B \, y, \, B \, z \\
B \, x & \rightarrow \text{left} \, x \, \text{null}, \, \text{right} \, x \, \text{null}
\end{align*}
\]
Extension to Interface Grammars

- In order to support shape types we extend the interface grammars as follows:
  - We allow *nonterminals with parameters*

- This extension is sufficient since the constraints about the connections among the objects can be stated using semantics predicates and semantic actions.
Doubly → p x, prev x null, L x
L x → next xy, prev yx, L y
L x → next x null

rule genDoubly(Node x) {
  «x = new Node(); x.setPrev(null);»
  apply genL(x);
}
rule genL(Node x) {
  choose {
    case:
      Node y = «new Node();»
      «x.setNext(y); y.setPrev(x);»
      apply genL(y);
    case:
      «x.setNext(null);»
  }
}
The use of shape types in interface grammars has two purposes

- For the objects that are passed as method arguments we need to check that their shape is allowed by the shape type
  - We call this *object validation*

- For the objects that are returned by the component we need to generate an object that is allowed by the shape type
  - We call this *object generation*
Object Generation vs. Validation

- Object generation and validation tasks are broadly symmetric
  - The set of nonterminals and productions used for object generation and validation are the same and are dictated by the shape type specification
  - In object generation semantic actions are used to set the fields of objects to appropriate values dictated by the shape type specification
  - In object validation these are constraints are checked using semantic predicates specified as guards
Object Generation vs. Validation

- There is a minor problem with object validation.
- In shape type specifications, the assumption is that there is no aliasing among the objects unless it is explicitly specified.
- This assumption is easy to enforce during object generation since every new statement creates a new object that has nothing else pointing to it.
- In order to enforce the same constraint during object validation, we need to make sure that there is no unspecified aliasing.
  - This can be enforced by using a hash-set for storing and propagating all the observed objects.
Experiments

- We wrote an interface grammar for the EJB 3.0 Persistence API
  - This is an API specification for mapping Java object graphs to a relational database
  - Hibernate is an implementation of this API
- Used several Hibernate test cases to evaluate performance and correctness
- Several test cases are designed to fail, and test exceptional behavior by violating the specification
- Accordingly we can verify the fidelity of our stub as well as verify the test cases themselves
## Verification Results

<table>
<thead>
<tr>
<th>Test case</th>
<th>Interface verification</th>
<th>Client verification</th>
<th>Err?</th>
</tr>
</thead>
<tbody>
<tr>
<td>bidir</td>
<td>2 s 15 MB</td>
<td>2 s 16 MB</td>
<td>no</td>
</tr>
<tr>
<td>mergeAndBidir</td>
<td>2 s 15 MB</td>
<td>2 s 16 MB</td>
<td>no</td>
</tr>
<tr>
<td>callbacks</td>
<td>2 s 15 MB</td>
<td>2 s 15 MB</td>
<td>no</td>
</tr>
<tr>
<td>exception</td>
<td>2 s 15 MB</td>
<td>2 s 15 MB</td>
<td>yes</td>
</tr>
<tr>
<td>clear</td>
<td>2 s 15 MB</td>
<td>2 s 15 MB</td>
<td>no</td>
</tr>
<tr>
<td>contains</td>
<td>3 s 26 MB</td>
<td>2 s 15 MB</td>
<td>yes</td>
</tr>
<tr>
<td>isOpen</td>
<td>2 s 15 MB</td>
<td>2 s 15 MB</td>
<td>no</td>
</tr>
<tr>
<td>persistNone</td>
<td>2 s 15 MB</td>
<td>2 s 15 MB</td>
<td>no</td>
</tr>
<tr>
<td>entityNotFound</td>
<td>2 s 15 MB</td>
<td>2 s 15 MB</td>
<td>yes</td>
</tr>
<tr>
<td>alwaysTransactional</td>
<td>2 s 15 MB</td>
<td>2 s 15 MB</td>
<td>yes</td>
</tr>
<tr>
<td>wrongId</td>
<td>2 s 15 MB</td>
<td>2 s 15 MB</td>
<td>yes</td>
</tr>
<tr>
<td>find</td>
<td>2 s 15 MB</td>
<td>2 s 15 MB</td>
<td>no</td>
</tr>
</tbody>
</table>
Discussion

• No test can run under JPF without an environment
• Verification is quite efficient
  — This is because the test clients are pretty small
  — The important thing is that we are able to reduce the state space by replacing the EJB code with our stub
• Relative to a hand written environment we do not seem to pay a speed or memory penalty
• Time taken to develop the interface was dominated by the need to understand EJB Persistence first; about a couple of hours
More Experiments

- We extended the interface specification to represent a recursive data structure for accounts and transactions.
- Accounts can have sub-accounts and, hence, are organized in a tree structure.
- We specified this tree structure in an interface grammar based on shape types and conducted experiments for verification of client code.

\[ \sum (\text{entry.amount}) = 0 \]
Four Clients

- We wrote 4 clients:
  - Client 1: Correct client, does not create any new data
  - Client 2: Correct client, creates new data
  - Client 3: Sometimes incorrect client
  - Client 4: Always incorrect client

- We increased the state space by increasing the number of accounts and entries and checked the verification performance
## Experiments

<table>
<thead>
<tr>
<th></th>
<th>Client 1</th>
<th></th>
<th>Client 2</th>
<th></th>
<th>Client 3</th>
<th></th>
<th>Client 4</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>MB</td>
<td>sec</td>
<td>MB</td>
<td>sec</td>
<td>MB</td>
<td>sec</td>
<td>MB</td>
<td>Acc.</td>
</tr>
<tr>
<td>0:11</td>
<td>26</td>
<td>0:17</td>
<td>27</td>
<td>0:10</td>
<td>27</td>
<td>0:14</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>0:14</td>
<td>26</td>
<td>0:23</td>
<td>37</td>
<td>0:16</td>
<td>36</td>
<td>0:13</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>0:21</td>
<td>34</td>
<td>0:38</td>
<td>39</td>
<td>0:20</td>
<td>36</td>
<td>0:14</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>0:49</td>
<td>36</td>
<td>2:55</td>
<td>41</td>
<td>0:17</td>
<td>36</td>
<td>0:14</td>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>3:38</td>
<td>36</td>
<td>15:37</td>
<td>50</td>
<td>0:18</td>
<td>36</td>
<td>0:14</td>
<td>27</td>
<td>1</td>
</tr>
</tbody>
</table>
Experiments

<table>
<thead>
<tr>
<th>Client 1</th>
<th>Client 2</th>
<th>Client 3</th>
<th>Client 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>sec</td>
<td>MB</td>
<td>sec</td>
<td>MB</td>
</tr>
<tr>
<td>0:14</td>
<td>26</td>
<td>0:23</td>
<td>37</td>
</tr>
<tr>
<td>1:09</td>
<td>35</td>
<td>2:35</td>
<td>41</td>
</tr>
<tr>
<td>19:09</td>
<td>37</td>
<td>34:18</td>
<td>43</td>
</tr>
</tbody>
</table>
Conclusions

• Modular verification is a necessity
• Interfaces are crucial for modular verification
• Interface grammars provide a new specification mechanism for interfaces
• We showed that interface grammars can be used for automated stub generation leading to modular verification
Related Work: Interfaces

- T. Ball and S. K. Rajamani. SLAM interface specification language.
- G. T. Leavens et al.: JML
Related: Grammar-based Testing

• A. G. Duncan, J. S. Hurchinson: Using attributed grammars to test designs and implementations
• P. M. Maurer: Generating test data with enhanced context free grammars
• P. M. Maurer: The design and implementation of a grammar-based data generator
• E. G. Sirer and B. N. Bershad: Using production grammars in software testing
THE END