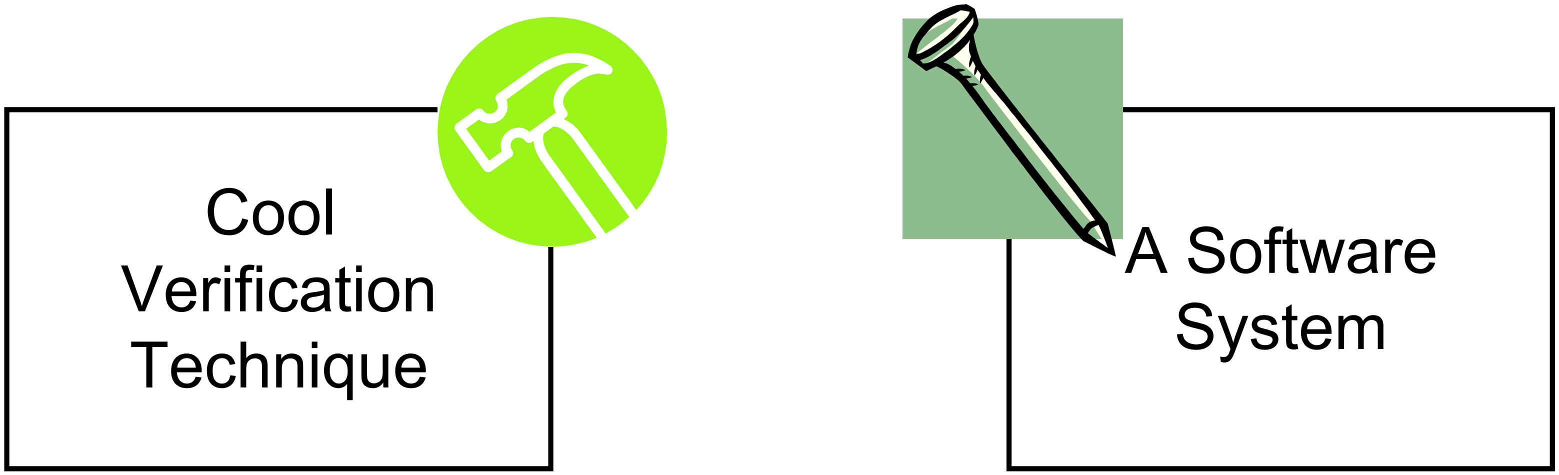
Extended Interface Grammars for Automated Stub Generation

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



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

Read-Write Lock in Action Language

```
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
```

Read-Write Lock in Java

```
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

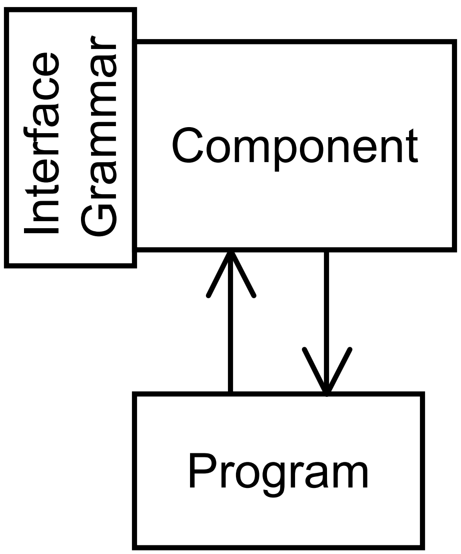
Interfaces for Modularity

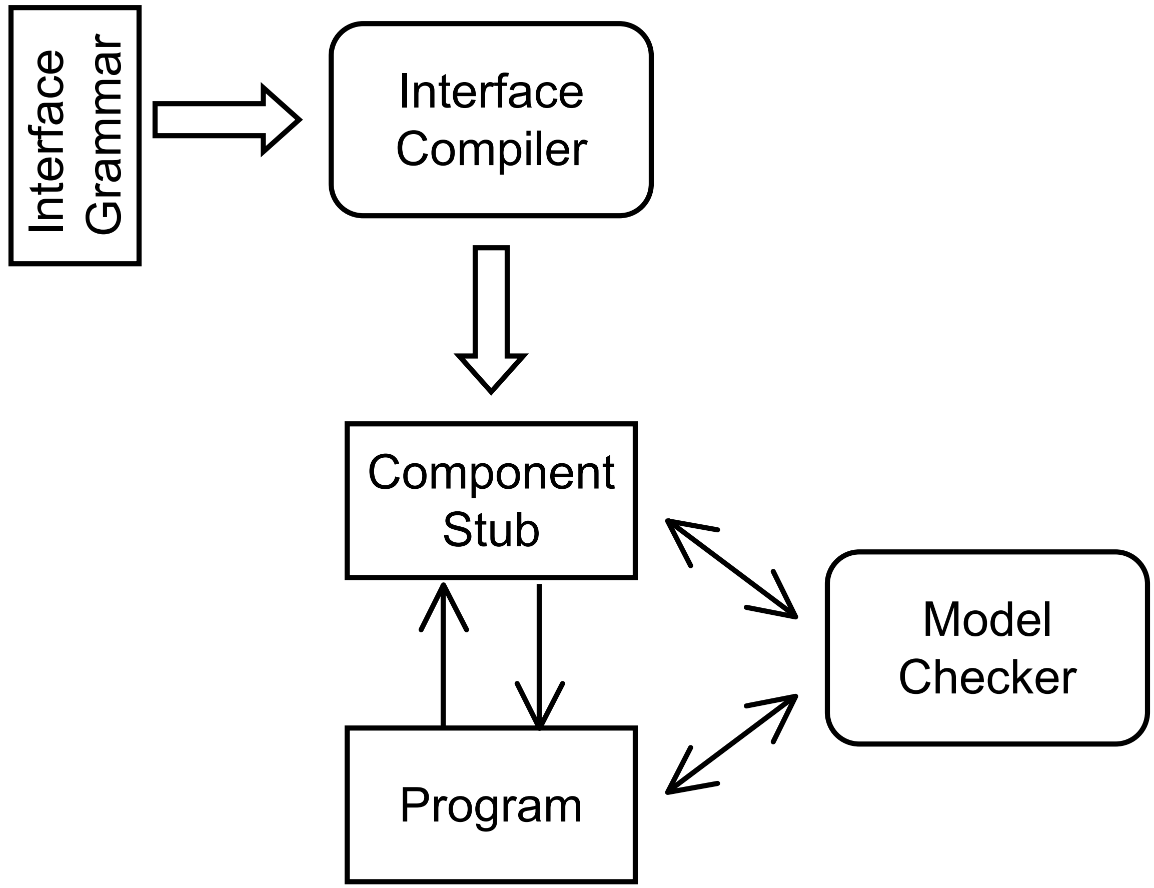
- How do we do ***modular verification***?
  - Divide the software to a set of modules
  - Check each module in isolation

- 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





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

`begin rollback begin commit`

- Here is a derivation:

$\mathit{Start} \Rightarrow \mathit{Base} \Rightarrow$ **`begin`** $\mathit{Tail}\ \mathit{Base}$

$\Rightarrow$ **`begin rollback`** Base

$\Rightarrow$ **`begin rollback begin`** $\mathit{Tail}\ \mathit{Base}$

$\Rightarrow$ **`begin rollback begin commit`** Base

$\Rightarrow$ **`begin rollback begin commit`**

Start	→	*Base*
Base	→	**`begin`** *Tail Base*
	|	ϵ
Tail	→	**`commit`**
	|	**`rollback`**

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

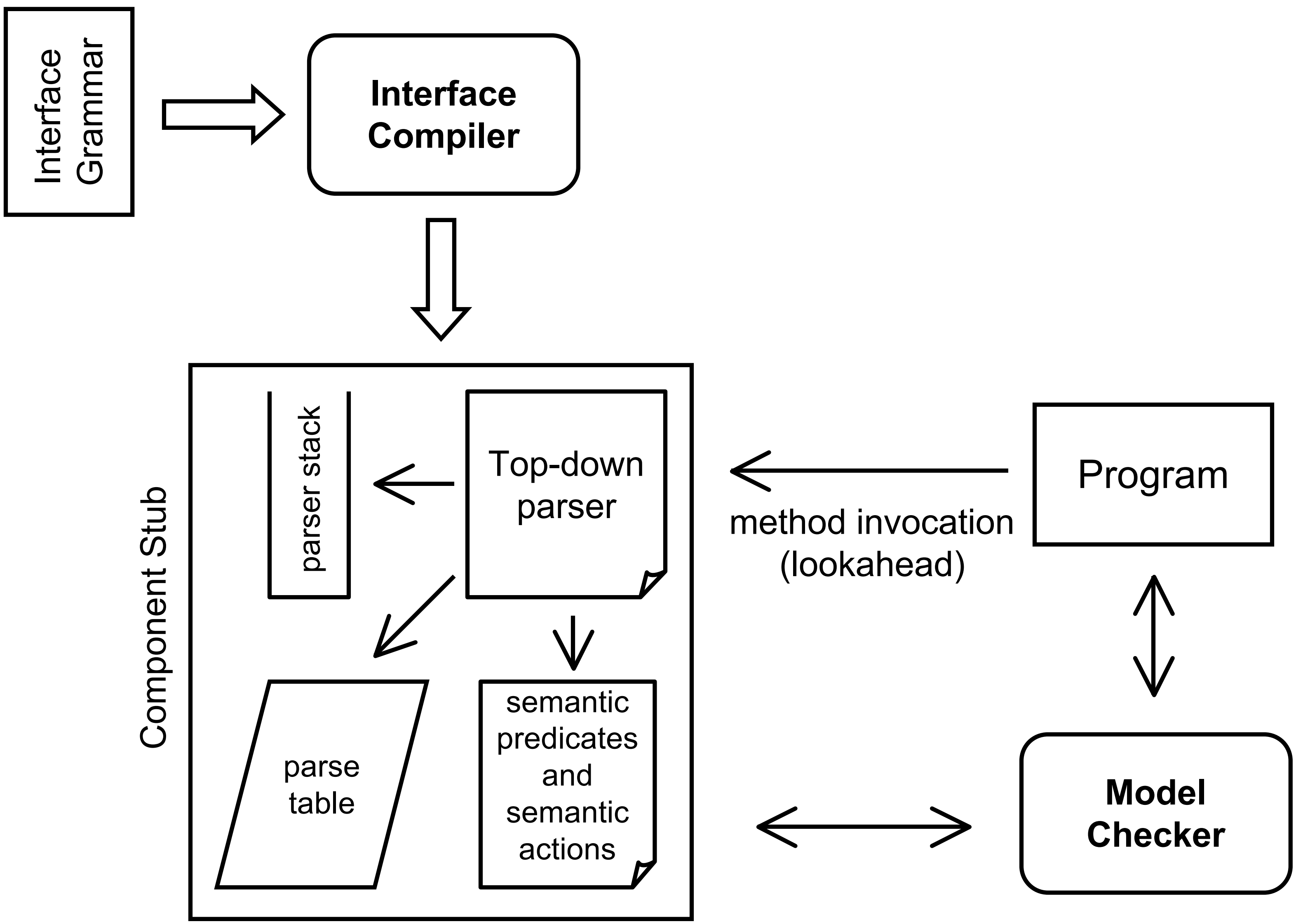
- Let's add another method called **`setrollbackonly`** which forces all the pending transactions to finish with **`rollback`** instead of **`commit`**
- We achieve this by extending the interface grammars with ***semantic predicates and semantic actions***

Start	→	«`r:=false; l:=0`» *Base*
Base	→	**`begin`** «`l:=l+1`» *Base Tail*
		«`l:=l-1; if l=0 then r:=false`» *Base*
	|	**`setrollbackonly`** «`r:=true`» *Base*
	|	ϵ
Tail	→	«`r=false`» **`commit`**
	|	**`rollback`**

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



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

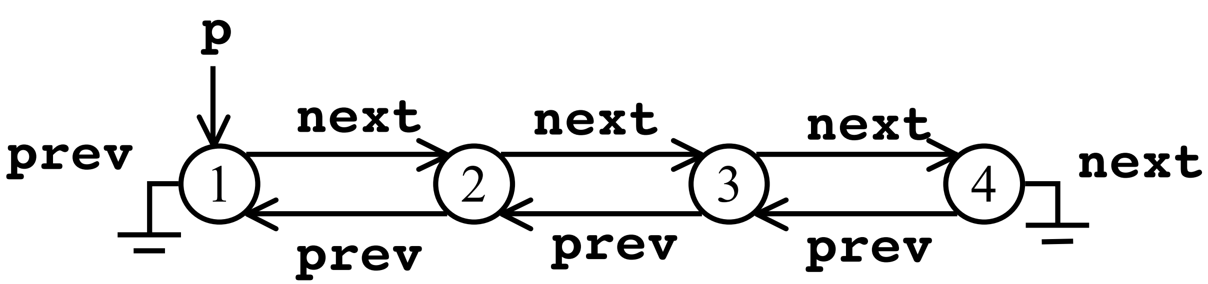
- Shape types [Fradet, Metayer, POPL 97] provide a formalism for specifying recursive data structures
- It is a specification formalism based on ***graph grammars***
- Shape types can be used to specify the connections among the heap allocated objects
- ***Objects*** become the ***parameters of the nonterminals*** 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

$$Doubly \rightarrow \mathbf{p}\ x,\ \mathbf{prev}\ x\ \mathtt{null},\ L\ x$$

$$L\ x \rightarrow \mathbf{next}\ x\ y,\ \mathbf{prev}\ y\ x,\ L\ y$$

$$L\ x \rightarrow \mathbf{next}\ x\ \mathtt{null}$$



$$Doubly \Rightarrow \mathbf{p}\ 1,\ \mathbf{prev}\ 1\ \mathtt{null},\ L\ 1$$

$$\Rightarrow \mathbf{next}\ 1\ 2,\ \mathbf{prev}\ 2\ 1,\ L\ 2$$

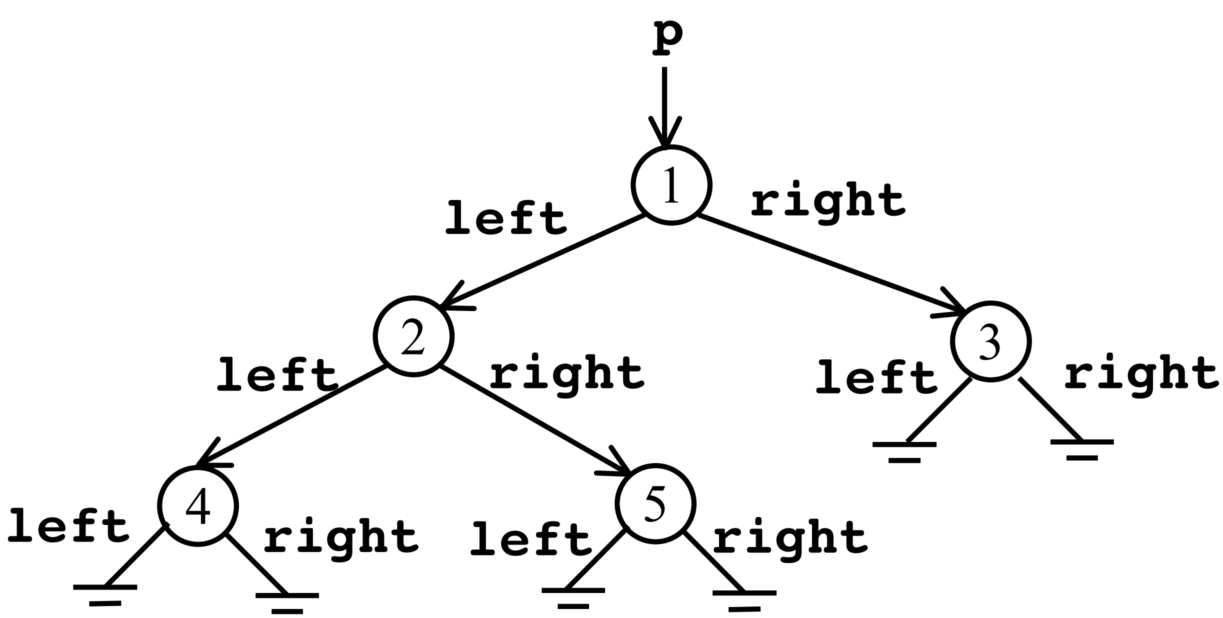
$$\Rightarrow \mathbf{next}\ 2\ 3,\ \mathbf{prev}\ 3\ 2,\ L\ 3$$

$$\Rightarrow \mathbf{next}\ 3\ 4,\ \mathbf{prev}\ 4\ 3,\ L\ 4$$

$$\Rightarrow \mathbf{next}\ 4\ \mathtt{null}$$

Shape Type for Binary Tree

Bintree	→	**`p`** *x*, *B x*
B x	→	**`left`** *x y*, **`right`** *x z*, *B y*, *B z*
B x	→	**`left`** *x* `null`, **`right`** *x* `null`



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

Interface Grammars + Shape Types

Doubly	→	**p** x, **prev** x null, $L\ x$
$L\ x$	→	**next** $x\ y$, **prev** $y\ x$, $L\ y$
$L\ x$	→	**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);»
}
```

Objection Generation vs. Validation

- 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

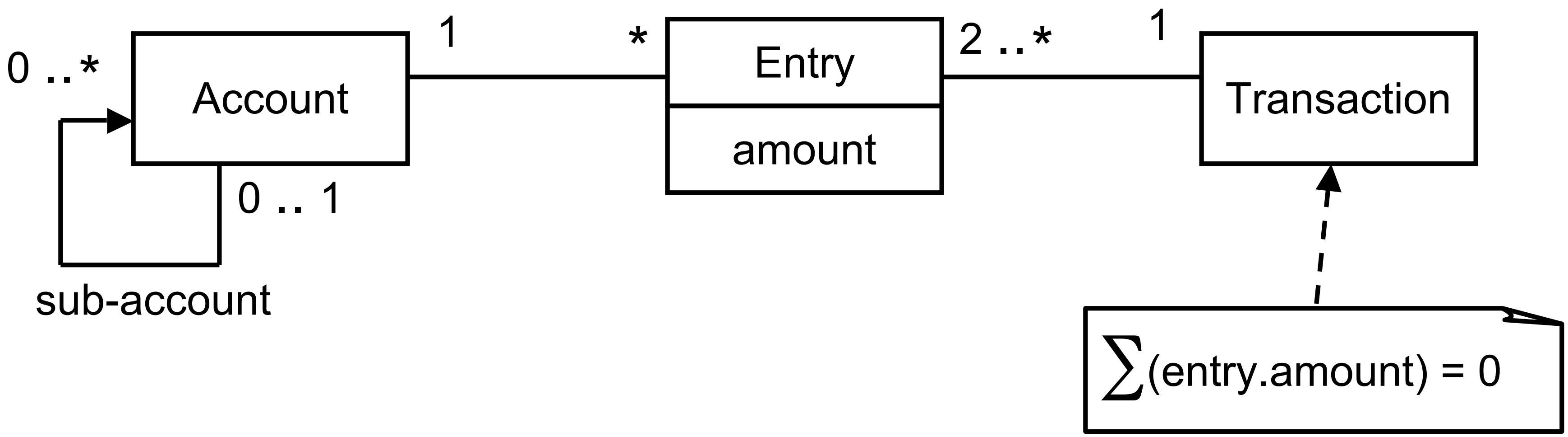
Test case	*Interface verification*		*Client verification*		*Err?*
bidir	2 s	15 MB	2 s	16 MB	no
mergeAndBidir	2 s	15 MB	2 s	16 MB	no
callbacks	2 s	15 MB	2 s	15 MB	no
exception	2 s	15 MB	2 s	15 MB	yes
clear	2 s	15 MB	2 s	15 MB	no
contains	3 s	26 MB	2 s	15 MB	yes
isOpen	2 s	15 MB	2 s	15 MB	no
persistNone	2 s	15 MB	2 s	15 MB	no
entityNotFound	2 s	15 MB	2 s	15 MB	yes
alwaysTransactional	2 s	15 MB	2 s	15 MB	yes
wrongId	2 s	15 MB	2 s	15 MB	yes
find	2 s	15 MB	2 s	15 MB	no

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-accunts 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



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

Client 1		Client 2		Client 3		Client 4			
sec	MB	sec	MB	sec	MB	sec	MB	Acc.	Ent.
0:11	26	0:17	27	0:10	27	0:14	27	1	2
0:14	26	0:23	37	0:16	36	0:13	27	1	4
0:21	34	0:38	39	0:20	36	0:14	27	1	6
0:49	36	2:55	41	0:17	36	0:14	27	1	8
3:38	36	15:37	50	0:18	36	0:14	27	1	10

Experiments

Client 1		Client 2		Client 3		Client 4			
sec	MB	sec	MB	sec	MB	sec	MB	Acc.	Ent.
0:14	26	0:23	37	0:16	36	0:13	27	1	4
1:09	35	2:35	41	0:56	38	0:13	27	2	4
19:09	37	34:18	43	14:03	39	0:19	27	3	4

Conclusions

- Modular verificaiton 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

- L. de Alfaro and T. A. Henzinger. Interface automata.
- O. Tkachuk, M. B. Dwyer, and C. Pasareanu. Automated environment generation for software model checking.
- A. Betin-Can and T. Bultan. Verifiable concurrent programming using concurrency controllers.
- 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