Interface-based Semi-automated Testing of Software Components

Tomas Potuzak, Richard Lipka, Premek Brada
Reliable Software Architectures (ReliSA) research group
Department of Computer Science and Engineering
Faculty of Applied Sciences
University of West Bohemia, Pilsen, Czech Republic

Introduction and Motivation I

- Component-based software engineering
  - An important trend in last two decades
  - Applications consisting of isolated parts called *components*
  - Components considered black boxes
  - Only interface visible, not the internal behavior of the component
  - Simplification of large applications development in teams
  - Better reusability and replacement of programmed functionality
  - Third party components, often with unavailable source code

Introduction and Motivation II

- Testing of software components within an application
  - Testing of correct cooperation of components, which cannot be accomplished by the creators of individual components
  - Often multiple versions of a single component
  - Regression testing desirable when a new version of a component installed into an application
- Approach for semi-automated regression testing of components with unavailable source code
  - Static analysis and runtime recording of the component’s behavior
  - Comparison of recordings of an old and a new version

Interface-based Component Testing I

- Designed for regression testing of a new version of a component within an application
  - Testing whether the new version of the component exhibits the same behavior as its old version
- Analysis of the behavior with the old component
  - Determining of interfaces, services, and methods of components
  - Generation of sets of invocations for the methods
  - Performing of the invocations, observing of consequences
  - Storing of the invocations and their consequences into a testing scenario file (XML)

Interface-based Component Testing II

- Analysis of the behavior with the new component
  - The old version of a component replaced by a new version
  - The same analysis performed
  - Another testing scenario file stored
- Comparison of both scenarios
  - The structures of both scenarios loaded and compared
  - All differences reported
- The entire application subject to the testing
  - Interactions of the components important to uncover their behavior

Invocation-Consequences Data Structure I

- All methods of all services of all components determined from the application
  - By any method capable of acquiring complete signatures
  - Stored into a tree data structure
- Initial set of invocation generated for all methods
  - Fully automatic in current implementation, but very simplistic (for each parameter several values, all combinations)
  - Additional values or restrictions provided by user when possible
  - Added to the tree data structure

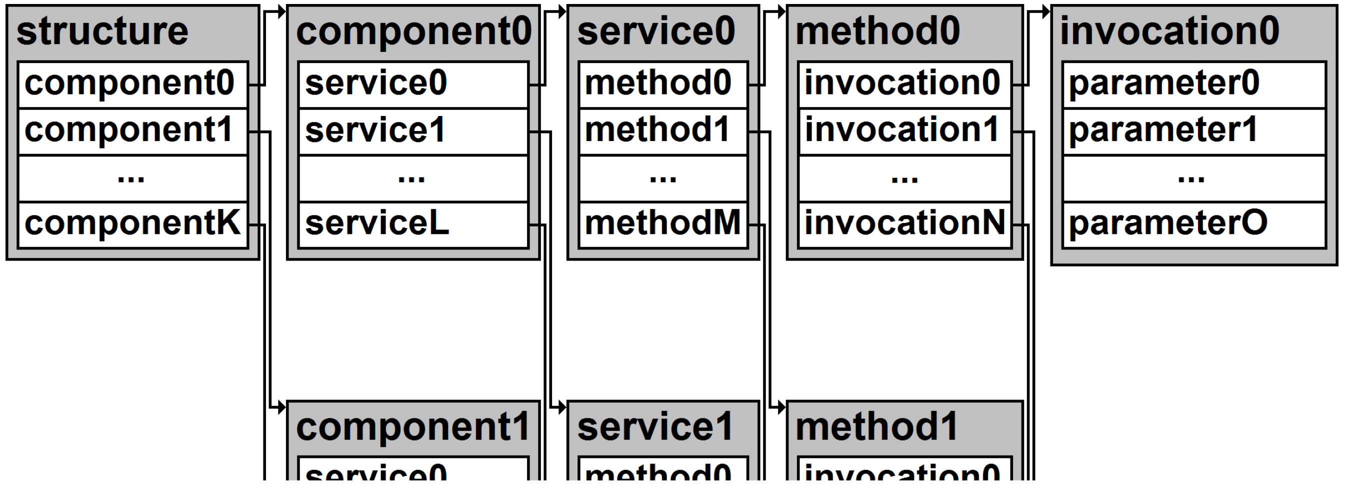
Invocation-Consequences Data Structure II

- Performing of invocations, observing consequences
- Possible consequences
  - The return of a value
  - A raised exception
  - A value change in “out” parameters of a method
  - A subsequent invocation of a service method of another component
  - Change of the inner state of the component – not easily observable, not considered
- Consequences added to corresponding invocations

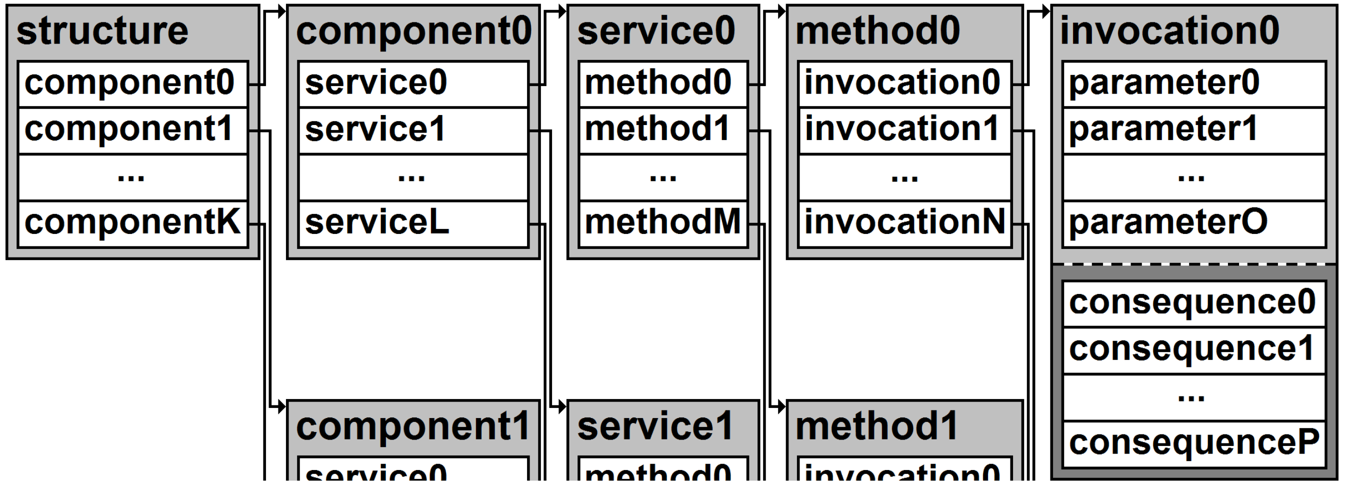
Invocation-Consequences Data Structure III

- Subsequent invocations
  - Most important
  - Invocations with “real” parameters, not generated
  - Invocations added into the tree data structure to the corresponding method
- The invocation-driven exploration of tree data structure repeated several times
  - To utilize the subsequent invocation from previous iteration for better analysis of the behavior
  - Stopped when no invocations and no consequences are added

Invocation-Consequences Data Structure IV



a) Generated tree data structure

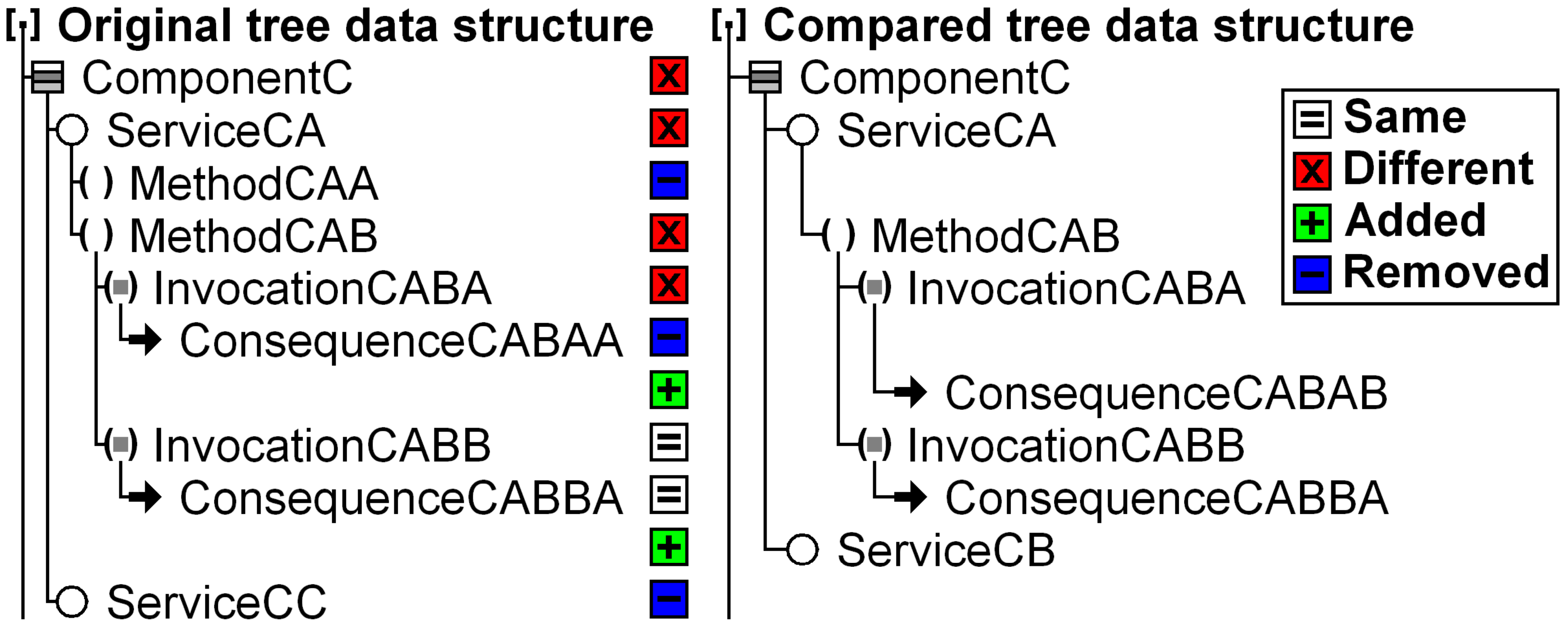


b) Added part after the exploration

Comparison of Tree Data Structures I

- Performed level by level
  - Components → Services → Methods → Invocations → Consequences
  - When an object is only in one tree data structure, this difference is reported and lower levels of this object are not explored
  - When an object is in both tree data structures, it is explored down to lower levels and the objects in lower levels are compared
  - The object is identical when all its sub-objects are identical
- Invocations and consequences levels important
  - Differences not detectable by static analysis

Comparison of Tree Data Structures II



Differences on the services level (ServiceCB added, ServiceCC re-moved, lower levels NOT compared), on methods level (MethodCAA removed), and on consequences level (ConsequenceCABAA repla-ced by ConsequenceCABAB)

Experimental Implementation

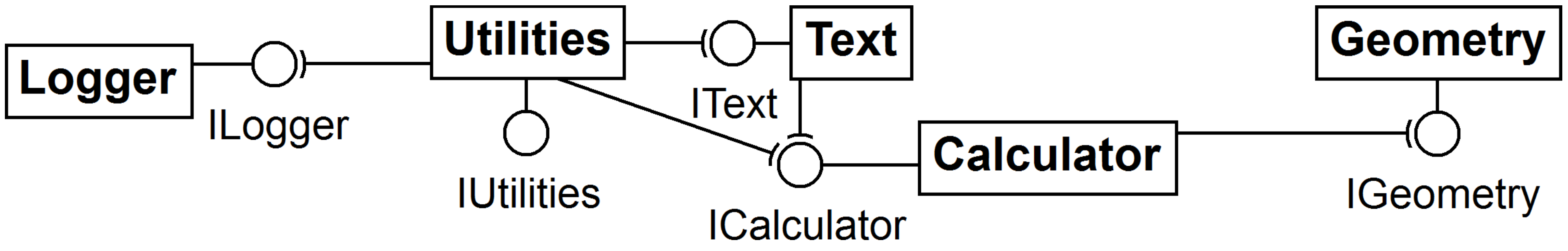
- Java & OSGi component model
  - Implemented as a single OSGi component (bundle)
  - Java reflection used for acquiring the signatures of methods
  - Generic proxy assigned to each service to intercept all method invocations using OSGi hooks
  - Predefined sets of values for each data type – all possible combinations used to generate the initial invocations

Validation and Results

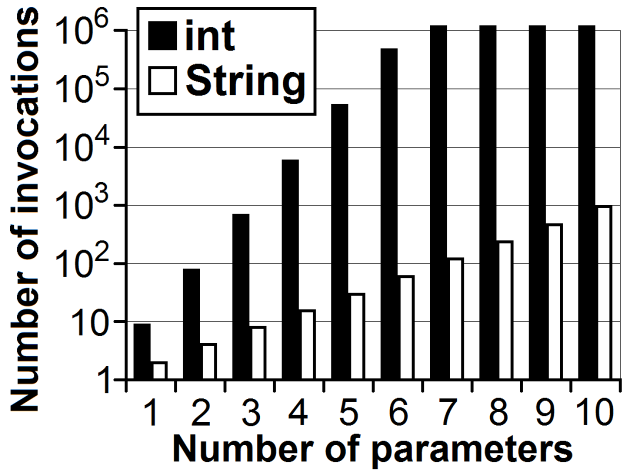
- Two sets of test
  - Dependency of the number of generated invocations on the number of components, methods, and their parameters
  - Ability to discover the differences in behavior when a new version of component is installed
- Testing environment
  - Standard desktop computer
  - Quad-core Intel i7-4770 CPU at 3.40 GHz, 16 GB of RAM
  - Windows 7 SP1 (64 bit), Java 1.6 (32 bit), Equinox OSGi framework

Number of Invocations Dependencies I

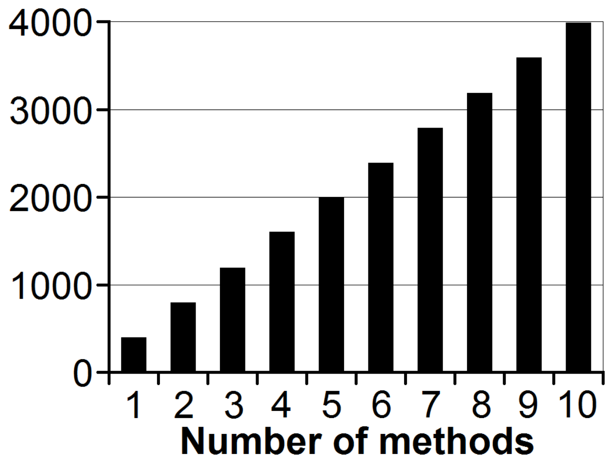
- Performed using a small application of our design
  - Enabling manipulation of the source code – necessary to perform the testing
  - A simple tool for mathematical calculations and string processing



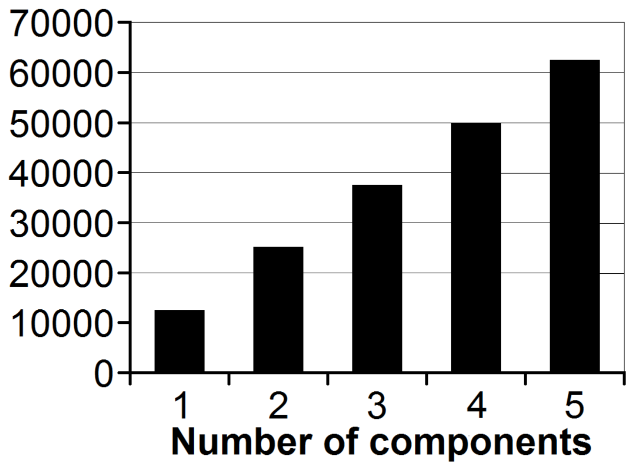
Number of Invocations Dependencies II



a) On the number of parame-ters of a single method



b) On the number of methods (5 parameters per method)



c) On the number of components (10 methods, 26 parameters in total per component)

- Methods with high number of parameters problematic
  - Quite rare
  - Better parameter generation planned

KATEDRA INFORMATIKY A VÝPOČETNÍ TECHNIKY

Ability to Discover Differences I

- Two applications
  - The simple application from previous set of test
  - CoCoME
- Three differences introduced (separately) into one component of each application
  - One method added – change #1
  - One method ceased to throw exception when invoked with *null* value – change #2
  - One method started to return *null* value instead of an instance – change #3

Ability to Discover Differences II

> Results for the test application

Structure	Original	#1	#2	#3
Explorations	3	3	3	3
Generated invocations	895	917	895	895
Subsequent invocations	747	769	569	725
Exceptions	10	10	9	10
Return values	910	933	890	910
Parameters changes	0	0	0	0
Differences (methods)	N/A	1	0	0
Differences (invocations)	N/A	0	21	0
Differences (consequences)	N/A	0	202	22

Ability to Discover Differences III

> Results for the CoCoME application

Structure	Original	#1	#2	#3
Explorations	2	2	2	2
Generated invocations	297	299	297	297
Subsequent invocations	0	0	0	0
Exceptions	224	224	213	224
Return values	1	3	12	1
Parameters changes	0	0	0	0
Differences (methods)	N/A	1	0	0
Differences (invocations)	N/A	0	0	0
Differences (consequences)	N/A	0	11	1

Conclusion and Future Work

- An approach for semi-automated regression testing designed for situations when a new version of component is installed into an application
  - The ability to discover differences demonstrated on two applications
- Future work
  - Improved generation of parameter values for method invocations
  - Work on automatic generator of complex data (i.e., objects and their collections)

Thank you for your attention

- Questions?