Deep Object Comparison for Interface-based Regression Testing of Software Components

Tomas Potuzak, Richard Lipka
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 with public interfaces
  - Third party components, often with unavailable source code, often multiple versions of a single component
- Semi-automated regression testing of components
  - Testing within an application with a new component version added
  - Static analysis and runtime recording of the component's behavior
  - Comparison of recordings of an old and a new version

Introduction and Motivation II

- Comparison of various objects during the testing process
  - Standard `equals()` method used in experimental implementation
  - Method not implemented at all, implemented incorrectly, or not considered all attributes values in some objects
- Deep object comparison
  - Created to mitigate the problems associated with `equals()` method
  - Objects compared based on the “shape” of internal structure and all corresponding primitive values

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

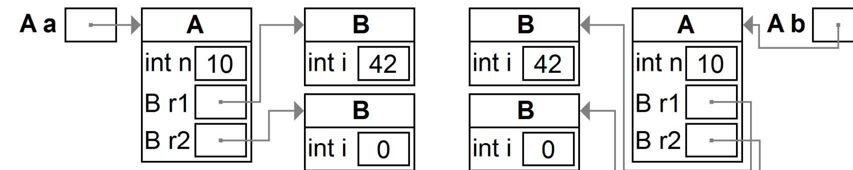
- Analysis of the behavior with the new component
  - The old version of a component replaced by a new version
  - The same analysis performed
  - Another testing scenario file stored
- Comparison of both scenarios
  - The structures of both scenarios loaded and compared
  - All differences reported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

Need for Deep Object Comparison

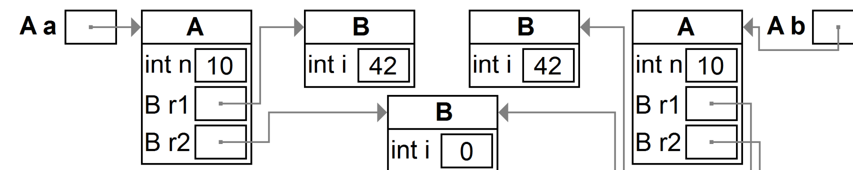
- General objects stored in both scenarios
  - For example return values, parameters, and exceptions
  - Source code unavailable → implementation of `equals()` not known
- Object comparisons performed
  - During construction of the scenario
  - During comparison of two scenarios
- Scenario stored in a XML file
  - Any structure utilized for object comparison must be able to be stored in a XML file

Deep Object Comparison I

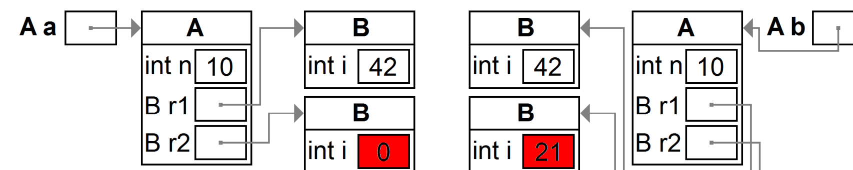
- Object equality
  - Objects of the same class
  - Graph representations of internal structures isomorphic
  - Same corresponding primitive values



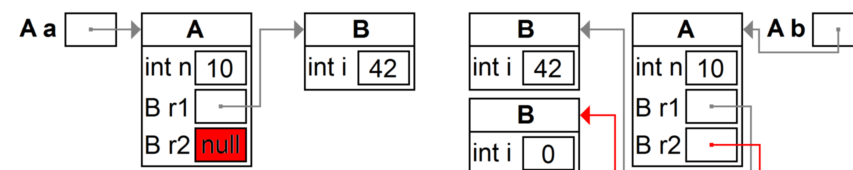
a) Objects considered equal - same shape, same primitive values



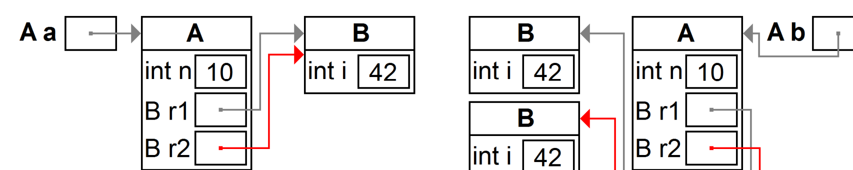
b) Objects considered equal - same shape, same primitive values



c) Objects considered different - different primitive values



d) Objects considered different - different shapes



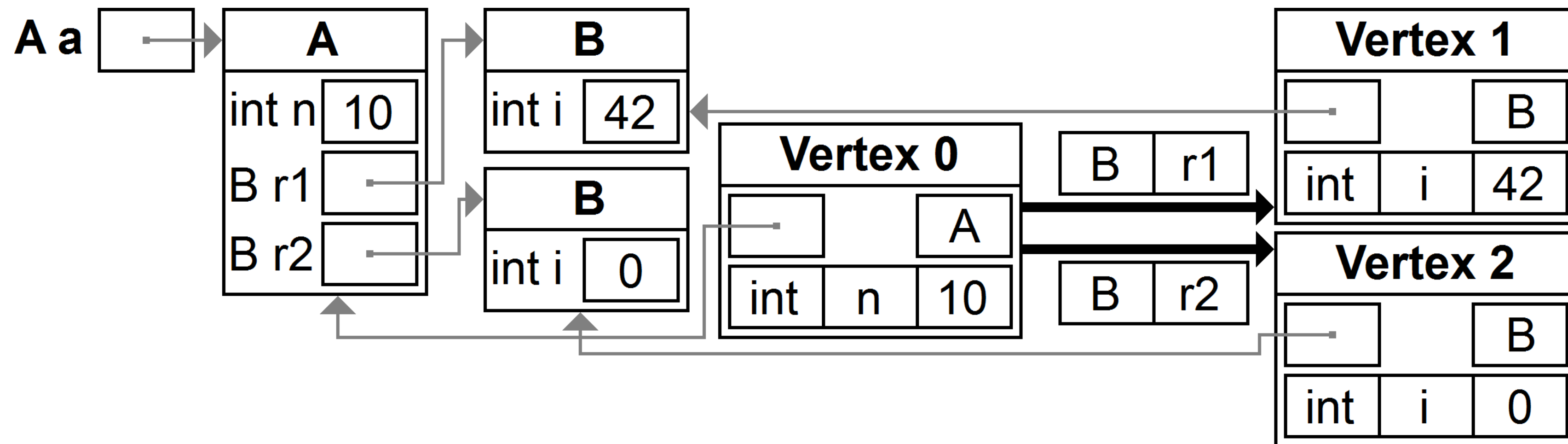
e) Objects considered different - different shapes

Deep Object Comparison II

- Forming of an object's graph
  - Directed vertex- and edge-labeled graph
  - Each vertex corresponding to an a single object of the internal structure of the compared object (with names, values and types of all primitive values and references to the object)
  - The starting vertex corresponding to the compared object
  - Each directed edge representing a reference attribute
  - Arrays treated as objects with their elements treated as variables
  - Breadth-First Search (BFS) of object internal structure for construction of the graph, list of visited objects to handle cycles

Deep Object Comparison III

- IDs assigned to all vertices
  - After graph creation
  - For storing of the graph to the XML file



Deep Object Comparison IV

- Comparison of two graphs
  - References to the original objects not used
  - Graphs containing all necessary values
  - Parallel BFS of both graphs
  - Both graphs explored in one loop
  - Checking of primitive values in corresponding vertices
  - Comparison ended prematurely due to difference in primitive values and/or missing corresponding vertex

Validation and Results I

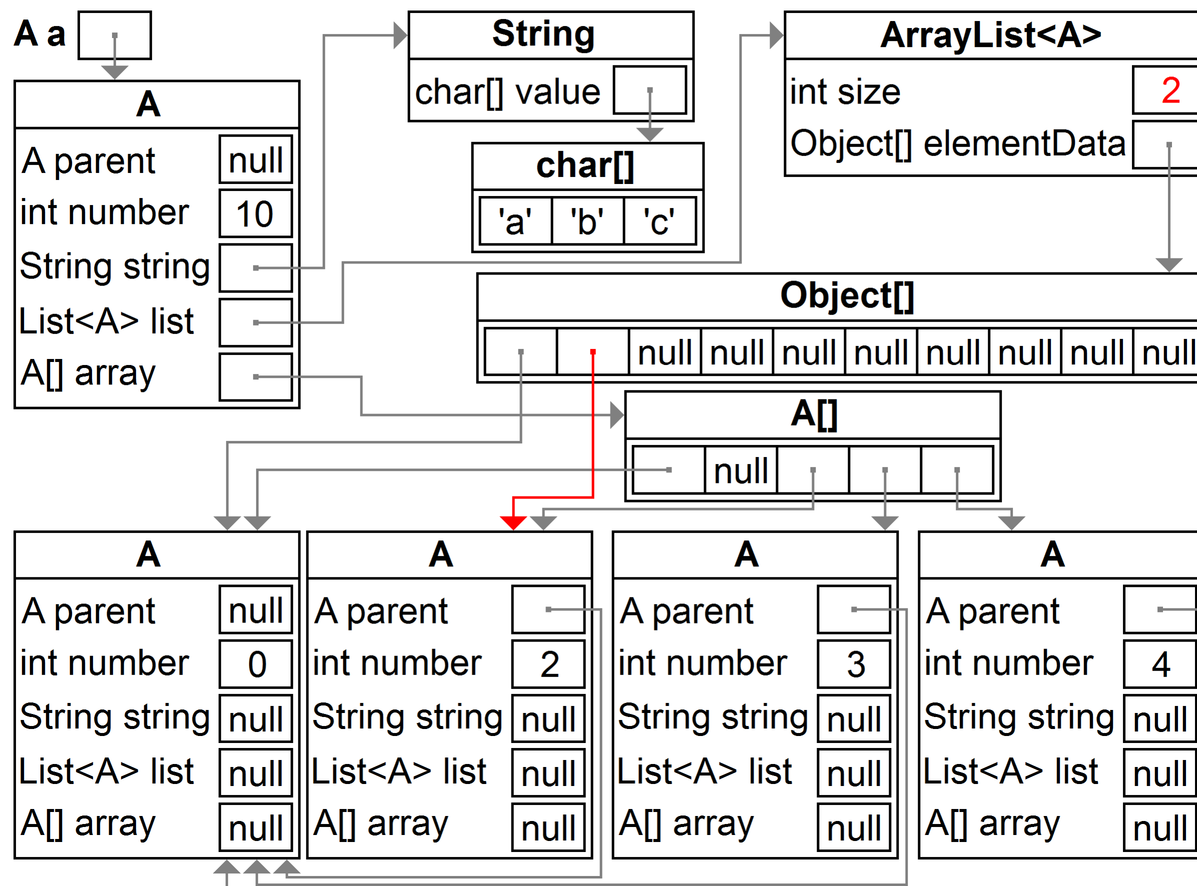
- Testing environment hardware
  - Standard notebook
  - Dual-core Intel i5-6200U at 2.30 GHz with HyperThreading
  - 8 GB of RAM
  - 250 GB SSD / 500 GB HDD
- Testing environment software
  - Windows 7 SP1 (64 bit)
  - Java 1.6 (32 bit)
  - Equinox OSGi framework

Validation and Results II

- Correct functionality of the algorithm
  - Comparison of pairs of similar or equal objects
  - 5 pairs with variously complicated internal structures
- Four tests performed for each pair
  - Both objects created in memory and compared (once equal, once similar, but slightly different)
  - Both objects created in memory, one stored to a file, loaded and then objects compared (once equal, once similar, but slightly different)

Validation and Results III

- Example of an object used for testing



Validation and Results IV

- Algorithm returning expected values in every instance

Pair of objects	Both objects in memory		One object saved & loaded	
	Equal	Different	Equal	Different
1	`true`	`false`	`true`	`false`
2	`true`	`false`	`true`	`false`
3	`true`	`false`	`true`	`false`
4	`true`	`false`	`true`	`false`
5	`true`	`false`	`true`	`false`

Validation and Results V

- Performance of the algorithm
  - Increasingly complex objects generated and compared
  - Only equal objects used for comparison – worst case scenario for the comparison time
- Graph construction phase and graph comparison phase times observed separately
  - Graph construction phase needed only once (during the scenario construction)
  - Graph comparison phase potentially needed many times (during the scenario construction and during the scenarios comparison)

Validation and Results VI

- Graph comparison times far lower than graph comparison times
  - Java reflection, sequential searching of visited vertices

Vertices count	Edges count	Graphs construction time [ms]	Graph comparison time [ms]
33	42	1.9	0.1
333	442	7.6	0.7
3 333	4 442	110.7	4.6
33 333	44 442	10 131.2	28.8

Conclusion and Future Work

- Deep object comparison described
  - Implemented, but not yet included into our interface-based regression testing
- Future work
  - Incorporate the deep object comparison into our interface-based regression testing
  - Utilization of the deep object comparison in another project focused on generation of complex testing data (comparison of generated objects and/or their parts to remove duplicity)

Thank you for your attention

- Questions?