DeepCompare: Static analysis for runtime software evolution

Yves Vandewoude
Yolande Berbers

Report CW 405, Februari 2005

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

DeepCompare: Static analysis for runtime software evolution

Yves Vandewoude
Yolande Berbers

Report CW405, Februari 2005

Department of Computer Science, K.U.Leuven

Abstract

Due to their modular design, component-based applications are relatively well-suited to supporting run-time evolution. However, replacing a component at run time without halting the application remains a difficult task. The main cause of this complexity can be found in transferring the state between two versions of a component. We have developed a methodology to perform run-time adaptations on component-based applications. This paper briefly introduces our methodology and then continues with a detailed study of our approach to state transfer. By exploiting the strong encapsulation of components, we conduct an analysis of the (object-oriented) source code of different versions of a component. A classification is presented which introduces different categories of change, according to the impact of the change on the set of objects that make up the component. We then describe different heuristics that are used to identify equivalent data structures between different versions. We show how these techniques are implemented in a tool called DeepCompare. An evaluation of our tool using two examples (an academic toy component and a real-life application), demonstrates that our process successfully identifies the vast majority of corresponding structures in a typical evolution scenario.

Keywords : Restructuring, Reverse Engineering, Reengineering.

DeepCompare: Static Analysis for Runtime Software Evolution

Yves Vandewoude*
Department of Computer Science
Celestijnenlaan 200A
Leuven, Belgium
yves.vandewoude@cs.kuleuven.ac.be

Yolande Berbers
Department of Computer Science
Celestijnenlaan 200A
Leuven, Belgium
yolande.berbers@cs.kuleuven.ac.be

ABSTRACT

Due to their modular design, component based applications are relatively well suited to support runtime evolution. However, replacing a component at runtime without halting the application remains a difficult task. The main cause of this complexity can be found in transferring the state between two versions of a component. We have developed a methodology to perform runtime adaptations on component-based applications. This paper shortly introduces our methodology, and then continues with a detailed study of our approach to state transfer. By exploiting the strong encapsulation of components, we conduct an analysis of the (object-oriented) source code of different versions of a component. A classification is presented which introduces different categories of changes, according to the impact of the change on the set of objects that make up the component. We then describe different heuristics that are used to identify equivalent data structures between different versions. We show how these techniques are implemented in a tool called DEEPCOMPARE . An evaluation of our tool on two examples (an academic toy-component and a real-live application), demonstrates that our process successfully identifies the vast majority of corresponding structures in a typical evolution scenario.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Restructuring, reverse engineering, and reengineering

1. INTRODUCTION

Research shows that over 80% of the cost of a software product is caused by maintenance, and more than 20% of the initial specifications of a product are considered outdated within a year after deployment ([24]). Keeping software up-to-date is a major problem that affects developers and users alike. The last few years, the tendency arose to address this issue by modularizing software with component-oriented methodologies. Applications are constructed by creating compositions of loosely coupled units of functionality: Components.

A large software system often consists of multiple interacting components. These components can be seen as large objects with a clear and well-defined task. Different definitions of a component exist; some see objects as components, while others define components as large parts of coherent code, intended to be reusable and highly documented. We base our definition on the one given by Szyperski [29, p. 41]:

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Szyperski's definition implies that all communication and data-exchange is explicitly specified in the interface of a component. As such, encapsulation of the internal structure of a component is absolute. Szyperski's definition does not, however, explicitly state how a component can be implemented. In fact, it is suggested that any programming technique (procedural, functional, assembly, . . .) is eligible. For the remainder of this paper, we assume that components are implemented using the object-oriented programming paradigm. Our component middleware assumes that components are written in the Java Programming language. In that case, a component usually consists of one or more classes that logically belong together as one unit of composition and deployment.

We further distinguish between a component instance and a component blueprint. The latter are reusable static entities that only exist at design time and contain a complete description of the type of a component and its implementation (the code). In addition, component blueprints have a unique identifier and a version number. A component instance is a runtime construct containing a state. For the remainder of the paper, we will reserve the term *component* for a certain component instance. If object oriented technology is used, a component can be considered as a tightly coupled group of objects, which we will call the *object-tree* of the component.

Generally speaking, components are composed by means of connectors. According to [1], a connector is a reusable design element that supports a particular style of component interactions. Our work assumes the interaction style that was defined in the SEESCOA project ([3, 31]). In this model, components communicate by asynchronously sending messages through external interfaces that are formally specified using ports. Connectors implement a pipe-like

*Supported by a scholarship Institute for the Promotion of Innovation through Science and Technology in Flanders

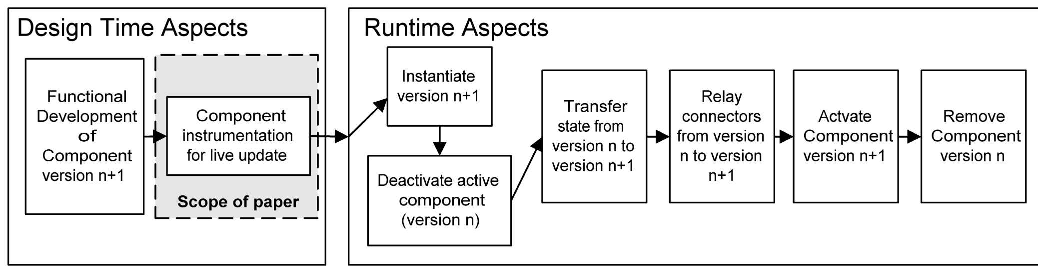


Figure 1: Different stages in the evolution process of a component. The scope of this paper is limited to the design phase.

construct, which makes relaying inter-component communication relatively easy to achieve at runtime. Most of the ideas and results that are introduced in this paper can, however, be used in the presence of a different communication paradigm.

Although reusing components shortens development time and increases robustness, the problems for the user of the application remain. Updating the software requires shutting it down and installing a new version, which is not always possible or desired. A solution for this problem can be found in *live updates*: modifying the software at runtime. Component-based software is relatively well suited for runtime adaptation due to its highly modularized design. Still, the replacement of one component by another remains a complex task. Most of this complexity is caused by the state which is contained in the active component. This state must be transferred to the new component version in order for the live update to succeed. Since the internal implementation of the component is changed between versions, the state may need to be transformed as well. The transfer and conversion of state-information needs to be implemented manually, a tedious and error-prone task.

In this paper we present an approach (backed by tool support) to assist a developer with the online replacement of a component. We begin with a general overview of our methodology and a situation of this paper in section 2 and continue with a detailed description of our approach to state transfer (section 3). Since not all changes have the same impact at runtime, a classification of three different changes is introduced in 4. Our work is implemented in a tool called DEEPCOMPARE[1]. The applicability of this tool is illustrated in section 5 were a small example is worked out in detail. In section 6, we evaluate the performance of DEEPCOMPARE on a larger project: an open source game. References to important related work are given in section 7. We discuss the limitations of our approach and present future work in section 8 and conclude this paper in section 9.

2. GLOBAL APPROACH

The problem of replacing a component at runtime can be split in two rather independent sub-problems: (i) the preparation of a component in advance (during the design of the component) and (ii) the online replacement process itself. Both deal with different issues in the context of evolution. In its entirety, the evolution of a component consists of a number of phases (see figure 1):

Designtime aspects:

1. *Functional Development:* First, the new component version must be developed. Ideally, the developer of the component is not concerned with problems related to the replacement process, and focuses only on the functional features of the component.

2. *Component instrumentation:* In this separate step, which can be performed after functional development of the new version has finished, the component is prepared for a live update. This entails both exposing the internals of the component state for future extraction (a task that can be automated completely), and the addition of functionality to import the state of its predecessor. That instrumentation process is the main focus of this paper.

Runtime aspects:

1. *Instantiation of the new and deactivation of the old version:* Upon deployment of the new component version into the running component system, the component runtime environment must instantiate a copy of this component in the runtime system[2]. The version of the component that was active is brought in a safe (quiescent: [19]) mode for replacement. The instantiation of the new component and the deactivation of the running version may occur in parallel, since the new component instance is not yet linked into the running application (all inter-component communication is routed through connectors).

2. *Execution of the state transfer:* At runtime, this activity involves nothing more than the passing of a reference of the old version to the new version and running the functionality present in the new component version. It is the responsibility of the new component to ensure that the transition is executed correctly.

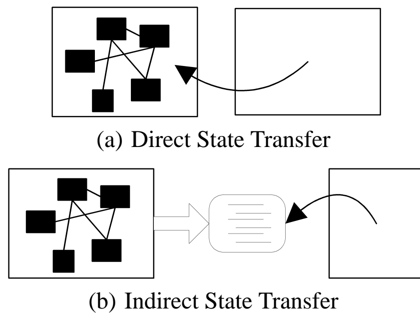
3. *Rewiring:* The component runtime must relink the connectors from the old version to the new component version. No messages may be lost in this process.

4. *Reactivation:* The new component is activated and starts processing messages (possibly queued by the underlying component system or connectors).

5. *Cleanup:* Optionally, the old component version may be removed from the component system to free up resources.

[1]The full source code of our tool is freely available for download at `http://www.cs.kuleuven.ac.be/~yvesv/dc/DeepCompare.zip`

[2]Since our components are implemented in Java, a mechanism of transparent class-renaming by a preprocessor is used to prevent class-reloading issues at runtime. We refer to [34, 35] for more information on the runtime aspects of our methodology.



(a) Direct State Transfer

(b) Indirect State Transfer

Figure 2: Two approaches to state transfer: either direct interpretation of the old implementation or indirectly through an exported abstract form.

Although our research tackles the evolution problem as a whole (eg, a runtime platform was developed to deal with the runtime aspects of component evolution), this paper exclusively deals with the evolution problem at design time.

The key-contribution of this paper is the design-time analysis-phase in which semantic information is derived from the component sources in order to determine which structures in different versions of a component are meant to correspond. The results from this analysis are embedded in the new component version through code-instrumentation. More specifically, semantic links between different types are stored in a hashtable which is used by the state transfer algorithm at runtime ([34]).

The other parts of our methodology and toolchain are discussed in [33] (a high-level overview of the runtime update process), [35] (a description of our component-middleware environment) and [34] (a detailed description of the runtime state transfer algorithm.)

3. APPROACH TO STATE TRANSFER

As mentioned previously, we assume components are implemented as a group of tightly coupled Java-objects. Intercomponent communication is explicit and asynchronous. Components are basically reactive entities that respond to incoming messages. The runtime environment is responsible for the management of all thread-related information and ensures that a component is placed in an inactive state before the actual replacement is initiated. Therefore, no methods are active when an update is initiated, and the entire state is contained in the instance variables of objects that make up its structure[3].

Two techniques exist to exchange state between versions:

Direct State Transfer: The implementation of the old version is used directly. It is the responsibility of the new version to interpret and convert the state from the previous version (see figure 2(a)).

Indirect State Transfer: The old version exports its state in an abstract representation which is later used by the new version (figure 2(b)).

Both techniques have their merit. The latter approach has the benefit that it allows easy upgrading when many different versions are involved. However, the technique strongly depends on the ability to construct an ontology that defines the abstract form. Therefore, this method is only used in specific and well defined fields such as protocol stacks (see [18]). Unless a global vocabulary is available and agreed upon that can be used to form an abstract state, any exported state is by no means more *abstract* than the one contained in the implementation of the component. As such, the main benefit of indirect state transfer (easy interpretation and import of the state by the new component version) is lost. Furthermore, the prerequisite that each component exports its own state to the abstract representation requires that this functionality is implemented by each component in advance, even if it is unsure whether the component will ever be replaced at runtime. Direct state transfer, albeit much more difficult to implement, does not suffer from these restrictions. Therefore, direct state transfer is more generally applicable and used in our methodology.

Since the new version is responsible for the import and conversion of the state of its predecessor, a method must be added that extracts this state and initializes the new component with that information. We aim to provide tool support that generates this method as much as possible. Achieving this goal requires solving sub-problems, which are fortunately relatively independent: identifying equivalent structures and generating the conversion routine itself.

3.1 Identifying equivalent structures

Before any form of state conversion can occur, one must first identify which member variables of both component versions are meant to correspond. It is clear that this process can never be completely automated. The main cause for this is that not all semantic information is present in the source code of a component. Therefore, it can not be extracted by a tool. A typical example is the representation of a triangle. In version n, three points may be used to represent its geometry. If the designer of the triangle decides to use two edges and an angle as the underlying representation of the triangle for version $n+1$, the relationship between the two structures can never be derived from the source. However, most evolution scenarios in which a live update is desired consist of relatively small changes. Components may be extended or modified, but the majority of their internal structure will remain the same. Using various heuristics (see section 5), a tool can identify most of these equivalent structures. We do emphasize however, that the identification of similarities is in essence an interactive and semi-automatic process. Complete automation is not attainable and not a goal of our work.

In practice, the problem of identifying equivalent structures in different versions of a component is tackled in three phases:

Construction of a source-model of both versions: The source code of both versions is parsed and a high-level model is constructed. While it is theoretically possible to conduct all further analysis on an abstract syntax tree, we consider such representations too low-level. The model is not only used to verify the correctness of the parsed code, it is also necessary for easy inspection and manipulation of the

[3]While this may seem a significant restriction at first, it makes good sense to allow methods to complete before an update is initiated. Pre-empting a method would either require rollback, or transferring state contained in the method-stack of the active method. Either technique would require substantial preparation (e.g. predefined interruption points would have to be identified). Even then, it is uncertain that such *equivalent points* exist (i.e., a location where the new method can continue where the old method left of).

source code. As our components are implemented in Java, existing Java-based meta-models can be used. A number of meta-models exist in literature, each with a different philosophy and complexity. At the time of writing, our tool uses JNome ([7]) as the underlying meta-model. Usage of the adapter pattern ([13]) ensures us that this underlying model is loosely coupled with the rest of our analysis framework.

Construction of similarity model: Every type from the old version is compared with every type from the new version, and the results of this comparison are stored in a *similarity model*. With its quadratic complexity ($O(n \times m)$, with n and m the number of types in the first and second version respectively), this is a rather expensive step. However, the number of types is usually not extremely high and the construction of the similarity model occurs during the preparation stage (and thus offline). The number of comparisons could be reduced drastically by using naming information of the types. However, this was deliberately avoided to defer all matching-logic to the next step. This way, certain refactorings (such as renaming a type or moving a type up or down the class-hierarchy) can be detected. For every couple of types, a detailed change description is constructed. This structure contains a description of the changes that are required to transform the first type into the second. Information such as the number of added, removed and changed members and interfaces, as well as the size (and therefore the probability that the two types under consideration are actually two versions of the same type) and type of the change according to the classification described in section 4 are present in the change description.

Change analysis of the similarity model: Finally, a detailed analysis is made of the similarity model that was constructed in the previous step. This is achieved by a number of *harvesters*, each of which encapsulates a specific matching-algorithm. Harvesters are linked in a chain and executed sequentially. Each harvester retrieves information from and stores new information into the similarity model by adding (or removing) *semantic links* between elements (types, member variables, methods, ...) of the first and the second version. As such, the similarity model represents all known similarities at each stage of the comparison process, and each harvester can benefit from collective knowledge and experience that was left behind in the similarity model by its predecessors. The selection of harvesters and their order can easily be configured by the user. Next to trivial matches (e.g. member variables with identical name and type in the same class are considered to be equivalent), more complex matches can be derived. We are currently investigating three categories of algorithms:

(i) *Structural Algorithms* will identify new similarities given known similarities and the structure of both components. This principle is used in the context of Database scheme evolution by Lerner ([21]).

(ii) Techniques from *Plagiarism Detectors* ([23, 38]) can be used to identify similar structures between different programs. The resulting similarities can either be directly exploited (e.g. similar variable names or comments used in the same context), or be used indirectly by other harvesters (e.g. by identifying similar methods – information that can be exploited by structural algorithms).

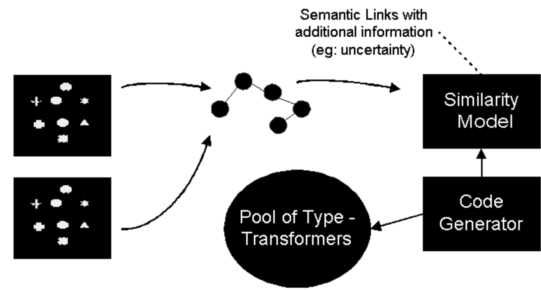


Figure 3: Schematic Overview of DEEPCOMPAREs entire analysis process.

(iii) *Refactoring Detectors* (e.g. [5]) exploit the principle that software evolves gradually. Due to the popularity of Extreme Programming ([2]), refactoring has gained much in importance in the field of reengineering. After a refactoring is found between different versions, underlying structure similarities can be identified.

Concrete examples of implemented harvesters will be introduced in section 5, where the applicability of our tool is illustrated on examples.

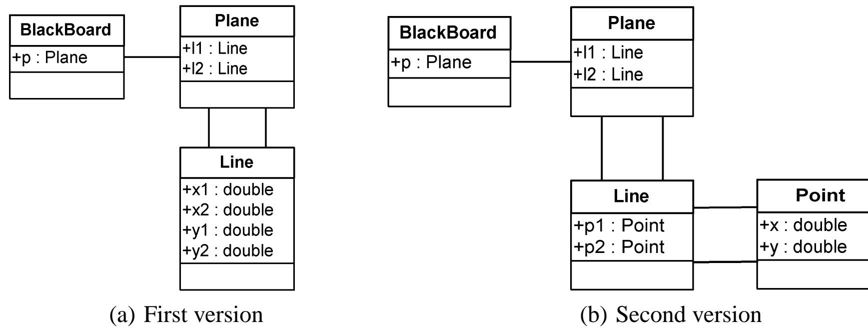
3.2 Generation of conversion code

After corresponding structures have been identified, code must be generated that converts structures from an old type to a new version. This is achieved by intelligently combining *type transformers*. A type transformer is the basic building block of the conversion process. Each transformer converts a structure of a given type into another type (e.g. a transformer may convert an array into a `Vector` or vice versa). Type transformers are implemented manually and added to a repository. Our tool DEEPCOMPARE will select transformers based on information present in the similarity model. Composite types are transformed by recursively applying known transformers to the different elements that make up the composite type. In addition, transitive application of known type transformers can be used when no direct type transformer is available. Since each type transformer is associated with a cost that represents the complexity of the transformation, an ideal conversion-path can be determined by using a shortest-path algorithm. Currently, DEEPCOMPARE offers a framework in which those type transformers can be used and combined.

An overview of the entire analysis process is shown in figure 3.

4. CHANGE IMPACT ANALYSIS

While many changes may be inflicted on a component at a given time, it is clear that not all of these changes have the same impact at runtime. Where possible, unchanged portions of the object-tree should be transferred by a simple reference copy instead of deep copying each object which requires transferring the state of these objects. In order to determine for which objects a simple reference copy suffices, a classification of changes is proposed. Depending on the category of a change, a different strategy can be used at runtime. Changes are classified during the construction of the similarity model and the result is attached to a change description (i.e.: for each type-pair, a change description with associated classification is constructed). Assuming that the implementation of a component consists of a number of interconnected objects, the classification of



(a) First version (b) Second version

Figure 4: Two versions of the BlackBoard component.

a change that was calculated for a type-pair relates to all objects of the class under consideration.

We will assume the following terminology for the remainder of this section. Types are referred to by capital letters, while objects of these types are identified by the lower-case version of the same letter. If two versions of a type are discussed, the subscripts $_{old}$ and $_{new}$ identify the version of the type. The class under consideration is identified by the letter `O`. Possible instance-variables of `o` (an object of class `O`) are named `m` (class `M`). We refer to the parent of `O` in the class-hierarchy as `P`.

Direct change: A change to `O` is *direct* if the change has a direct impact on the class (and therefore its instances) itself. The following changes to a class are considered to be direct:

- The addition or removal of a method or member variable to or from `O`.
- A change in the signature or implementation of a method in `O`.
- A change in the signature of a member variable (i.e. changing the visibility, adding a modifier, changing the type, . . .) of `O`.

Summarized, every change that results in a change in the sourcecode of `O` is marked as a direct change. The idea behind this category is that the impact of such a change on the representation of an object at runtime is of such a nature, that it is not possible to reuse the object in the object-tree that makes up the new version of the component. As such, changes in the inheritance hierarchy of the class to which the object belongs also qualify as direct changes (i.e. `O` is automatically considered to be directly changed if its parent `P` is directly changed).

Indirect change: A change to an object is considered to be *indirect* if the object itself does not change, but the type of one of its member variables does. Therefore, `O` is indirectly changed if the type `M` of one of the instance variables of `o` is directly changed. A change in the implementation or structure of `M` clearly has implications on the behaviour of `o` itself. However, at runtime, the impact of this change is limited with regard to the structure of `o`. Only a reference to the object represented by the changed member variable is stored in `o`. Therefore, it suffices to relay a pointer at runtime to make sure that `m` links to an implementation of $\mathtt{M}_{new}$ in order to incorporate the change into the old object. `o` itself can in theory be reused and must not be replaced from the object-tree that makes up the component. While it is certainly true that this pointer-relaying to an object of a different type ($\mathtt{M}_{new}$) than the type which was declared at compile time (`M`) is not supported by most virtual machines, its implementation is not extremely difficult. Also, while such pointer-relaying may violate type safety in the general case, this is not an issue with indirect changes.

Assuming that both versions of the component are semantically and syntactically correct by themselves, changes in the behaviour of its member variables usually have no impact on `o`: methods or member variables that may have been introduced in $\mathtt{M}_{new}$ are not used by `o`. If they were, `O` would have been changed directly itself. Methods or member variables that were removed from `M` were not used in the implementation of `O` in the first place, since the new version of the component would not be type sound if this were the case. Those changes that actually do have impact on the behaviour of `o` are desired by the programmer of the new component version and have no impact whatsoever on the type soundness of the new composition (e.g. a bug fix in `M` will immediately be used by `o` after a relaying of its member variable.

Influenced : An object is *influenced* if it has associations to types that are indirectly changed or influenced themselves. While this category reflects the possibility of a behavioural change of the influenced object, the change itself was not caused by a change in its implementation. At runtime, nothing needs to be done and an influenced object can be reused without any change in the new component version.

In order to further clarify these three terms, let us consider the small example shown in figure 4. In this example, a `BlackBoard` is shown. Its implementation uses a predefined class: `Plane`, which in turn is defined by two intersecting lines whose type is defined in the class `Line`. The author of `Line` has decided that its current implementation (containing 2 `x`-coordinates and 2 `y`-coordinates) is not ideal, and introduces an additional class (`Point`) to further encapsulate coordinate calculations (see figure 4(b)). The change is entirely contained within the boundaries of the class `Line` and no changes were necessary to either `Plane` or `BlackBoard` to compile the new version.

In this case, a direct change was made to `Line`. No changes were made to the class `Plane`. However, objects of the latter must be changed at runtime, since references to objects of the old `Line` im-

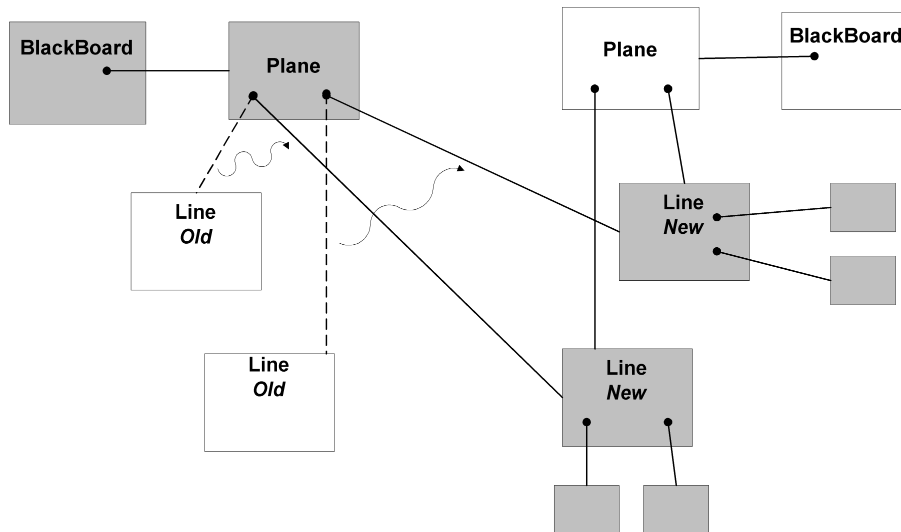


Figure 5: Runtime transition between two versions of the `BlackBoard` component. Grey objects remain present in the final object-tree of the new component version.

plementation are present. The change is therefore indirect. Nevertheless, assuming some virtual machine support, objects of `Plane` can be reused if the associations to the new line objects are updated. The `Blackboard` is only influenced. Possible behavioural changes are desired and the object remains entirely unchanged at runtime. Therefore, it is not even to be considered during state transfer. The effect of the transition between these two versions is shown in figure 5. Grey objects are present in the runtime version of the new version after the replacement has completed. The resulting component consists of a hybrid object-tree in which both old and new objects are present.

The distinction between direct, indirect and influenced changes is important since they allow us to keep a portion of the old object-hierarchy in place. Since no new versions of these objects need to be constructed, and no state transfer must be initiated, this speeds up the actual evolution drastically. The effect of this phenomenon is directly proportional to the size of the object-tree and more distinct as fewer classes are changed.

5. THE DEEPCOMPARE TOOL

The methodology that was introduced in previous sections has been implemented in a tool: DEEPCOMPARE . It requires the source code of both component versions as its input, and returns a detailed description of the correspondences found between those versions. The output is available both graphically (which is convenient for smaller applications and to spot trends), and in textual form (an XML file is generated which contains a detailed description of all generated information), which is preferred for larger projects and for automatic processing by tools later on. This section first provides more concrete information concerning implementation aspects, such as the harvester configuration that was used to test the application. We explain the algorithms of the harvesters using the example introduced in section 4. A more realistic case is worked out in section 6.

5.1 Construction of the similarity model: Calculating diffs for all type-pairs.

After both versions of the component are parsed and a model is constructed, a change description is constructed for each pair (a, b) where a and b are types from the old and new version of the component respectively. Such a change description is basically an enumeration of all the differences between the two types. In addition, a change description also specifies the classification of the change. Since the construction of such a change description is a computationally expensive operation, and since most harvesters (see section 5.2) will use this information in their decision making process later on, the in-advance construction of these change descriptions for each type-pair drastically reduces runtime complexity.

In our example, the change description of the couple (`BlackBoard`, `Point`) would result in a classification `Direct` since the implementation of both types does not correspond. The calculation of a change description is straight-forward. Even for two entirely unrelated types such a change description can be constructed: one simply removes all methods, variables and interfaces from the first type, and subsequently adds all methods, variables and interfaces of the second. The main challenge that is addressed at this stage is deciding whether a single member variable or method was changed (e.g. renamed) or whether a variable was removed and a different and unrelated one was added.

Following heuristics are used to construct the change description (see figure 6):

(i) The fully qualified name of both types is checked for differences. If the type is known to belong to an external library (e.g. a class from the JDK), nothing more is required, since library types are considered not to change[4].

(ii) Two member variables are considered to be equal if their

[4]This does not restrict the generality of our approach: if a used library does change between versions, it must be parsed as part of the component sources.

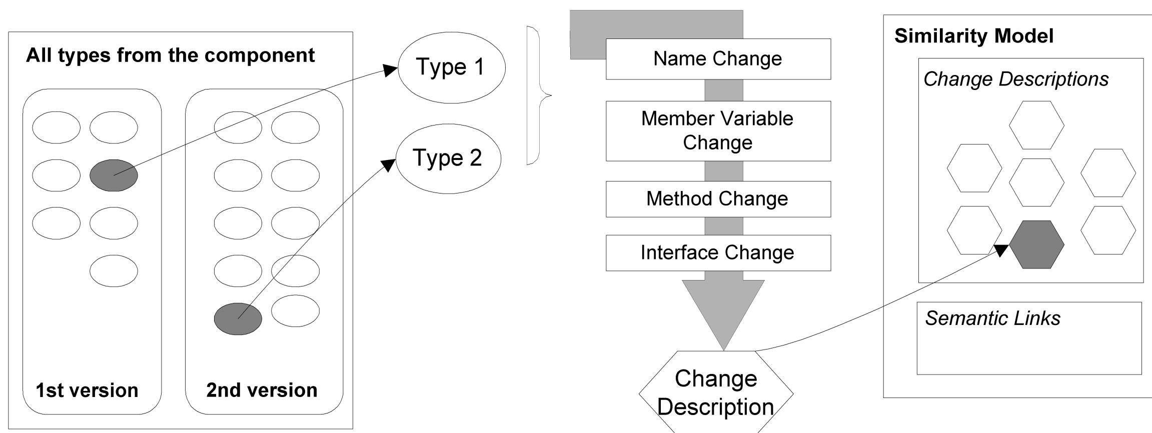


Figure 6: During the construction of the similarity model, detailed descriptions of changes are calculated for each type-pair. No semantic links have been added to the similarity model so far, since these are added by the harvesters at a later stage.

type, name and modifiers match. Two methods are said to be unchanged if their signature and implementation[5] matches.

(iii) Two member variables are considered to be *changed* if their type matches, none of them is already involved in an equivalence relationship (see (ii)) and their name is sufficiently similar (we implemented fuzzy matching on names using a variant of the Greedy String Tiling algorithm as described in: [37]). Methods are considered to be changed if they are not involved in an equivalence relation and their signature is equal or similar (i.e.: same return type and a similarity on the name which exceeds 75%).

(iv) Members are considered to be removed if they are present in the first version and no match is found with a type from the second version. Members are considered to be added, if they are present in the second version, and no match can be found with a type in the first version of the component.

All these change descriptions are stored in the similaritymodel. We wish to stress once more that no verdict is made on the similarity between two types. By constructing a change description for every pair of types, a high-level description of changes can be offered to the harvesters. The similarity model also offers facilities to add or remove semantic links and to query the model for information.

5.2 The harvesters: What corresponds?

It is the task of the harvesters to create semantic links in the similarity model. Different harvesters are executed sequentially. The selection of the harvesters and the order of their execution can easily be selected by the user of DEEPCOMPARE. Each harvester can execute queries on the similarity model to retrieve both change descriptions and semantic links that were added by its predecessors. A harvester will then, depending on its own decision logic, decide whether or not to add links to (or remove links from) the similarity model. Although none of the harvesters that are currently implemented exploit the possibility to remove semantic links from the similarity model, the option is available. For instance, it could be used to implement a harvester which detects known false positives generated by one of its predecessors. After execution of the harvester chain, the similarity model contains a detailed description of corresponding elements between the two versions of the component.

[5] Currently, implementations are considered to be identical if their source code is identical. In the future, more complex comparisons can be used that would allow some types of changes that have no effect on the outcome of the method (such as variable renaming, converting a `for`-loop to a `while`-structure, ...

The chain of harvesters which was used to test the performance of DEEPCOMPARE is shown in figure 7.

IndenticalClassesHarvester: This harvester will iterate over the change descriptions in the similarity model and will establish a semantic link between all types that are not directly changed.

SimilarClassesHarvester: This harvester will iterate over all types who have remained unlinked by the previous harvester. The SimilarClassesHarvester will assign an impact-rating to each change which is based on characteristics of the change description:

$$\begin{aligned} impact \;=\; & \alpha * sameName? + \\ & \beta * nrOfChangedMethods + \\ & \gamma * nrOfChangedMemberVariables + \\ & \delta * nrOfIdenticalMethods + \\ & \epsilon * nrOfIdenticalMembers + \\ & \zeta * !changedNrOfMembers? + \\ & \eta * parentalImpact \end{aligned}$$

`sameName?` and `!changedNrOfMembers?` are integer representations (0 or 1) of boolean expressions that indicate whether the name of the type has changed and whether the number of methods/member variables remained the same. The SimilarClassesHarvester will establish a semantic link between two types if the impact of their change is higher than a predefined cutoff value. Both the cutoff value and the value of the different constants were empirically chosen to yield a maximum amount of matches while minimizing false positives.

IdenticalClassVariableHarvester: Unlike the two first harvesters, this harvester will focus on identifying similarities

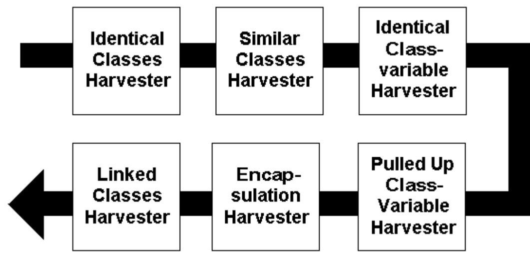


Figure 7: The harvester chain used for the case described in this paper.

between member-variables, and not between types. It will iterate over all types that have been linked by the two first harvester. It will then add semantic links between member variables that are identified as similar or identical by the change description.

PulledUpClassVariableHarvester: This harvester searches for unlinked member variables that belong to type $T1$ of the first version which has been linked to a type $T2$ of the new version. While the variable may very well have been removed, this harvester starts from the assumption that it is moved up or down the class hierarchy. It will therefore search for unmatched member variables in the parent-class or the children of $T2$. In our example, this harvester will assume that the 4 integers that make up the coordinates of a `Line` are moved up or down the hierarchy. Since no suitable candidates are found however (the parent of `Line` is `java.lang.Object` and as such, unchanged), the harvester is unable to find matches in our example.

EncapsulationHarvester: Similar to its predecessor, this harvester takes as a starting point that it is more likely for a variable to have moved than to have disappeared. However, this harvester assumes that the member variable was encapsulated. It will therefore search for unmatched variables inside the type of unmatched variables of $T2$. In our example, this harvester will assume that the 4 integers that make up the coordinates of a `Line` are encapsulated. Since it locates unmatched member variables in the new version of `Line` (`p1` and `p2`), it will search the class `Point` for possible matches for these coordinates, and will subsequently establish a link between the coordinates of a `Point` and the coordinates of a `Line`.

Linked Classes Harvester: This harvester will search for unmatched types that contain member-variables that have been matched. Since it is unlikely that two totally unrelated types do have member-variables in common, this harvester will perform an additional analysis between those types, similar to SimilarClassesHarvester.

Figure 8 shows a screenshot of our tool. All types from the first version are shown on the left, whereas types from the new version are shown on the right. Instance variables (which encapsulate the state) are shown as ovals surrounding the type to which they belong. Although this is not visible in a black and white printout of this figure, colors are used to distinguish between user defined types (red for the first version, blue for the second version), library types (such as the JDK) which are shown in yellow and primitive types that are shown in orange. The figure shows that the results are more than satisfactory. All types are correctly identified, and even the encapsulation using the `Point` type was nicely detected. However, it is clear that the selection and order of the harvesters determines the success of the comparison. The default toolchain of our tool generates nice results. Nevertheless, the creation of a custom chain is easy to do and can increase the accuracy of our tool. For instance, if the developer has used a given refactoring, he may include a harvester which correctly detects the changes that were implemented. In addition, our tool allows for the developer to add or remove semantic links manually to the similarity model. These human actions are marked and can never be corrected by harvesters, which allows a developer to guide the detection process. So far, we have not yet needed this functionality however.

6. EVALUATION

While the example from section 5 nicely illustrates the possibilities of our approach, it is hardly a realistic scenario. An ideal test platform for our tool and methodology had to meet a number of important requirements:

Realism: In order for a testbed to offer some realistic results, it is vital that the testbed is not specifically designed and programmed with DEEPCOMPARE in mind. It is our goal to minimize inevitable bias that would occur if a test case is specifically constructed for a methodology.

Size: It is clear that the size of the test should be larger than the academic toy-example discussed in the previous section. The target of our methodology is component-evolution. Therefore, the size of our target should more or less correspond with the size of a real-life component implementation. However, it must also be feasible to manually verify the results that are generated by the comparison process.

Availability: Since the methodology and tool deal with *evolution*, many different versions of the test must be available. It is preferred that the code of the test is available to the general public since this allows for reproduction of our results by the reader.

Familiarity with code: For practical reasons, it is desired that the authors have some familiarity with the code of the test. This both eases the execution and increases the reliability of the (manual) verification of DEEPCOMPAREs results.

Java-Based Component: Ideally, the test should be in the form of a component: a unit of deployment with strictly defined external interfaces. Additionally, since our tool only accepts Java as the source language, the test must be implemented in Java.

Considering all these requirements, an open source game (named JTeg) was chosen. JTeg is a Java-based client for a client-server risk-clone that was developed by one of the authors as a hobby project a few years ago. Its age guarantees the realism of the scenario, since there is obviously no design bias towards DEEPCOMPARE or software evolution in general. Being codesigned by one of the authors of this paper, there is a familiarity with the code. In addition, the game is hosted on Sourceforge which ensures public availability of many different versions. After all, the Sourceforge CVS repositories have public read access to the general public. Depending on the version considered, the size of JTeg

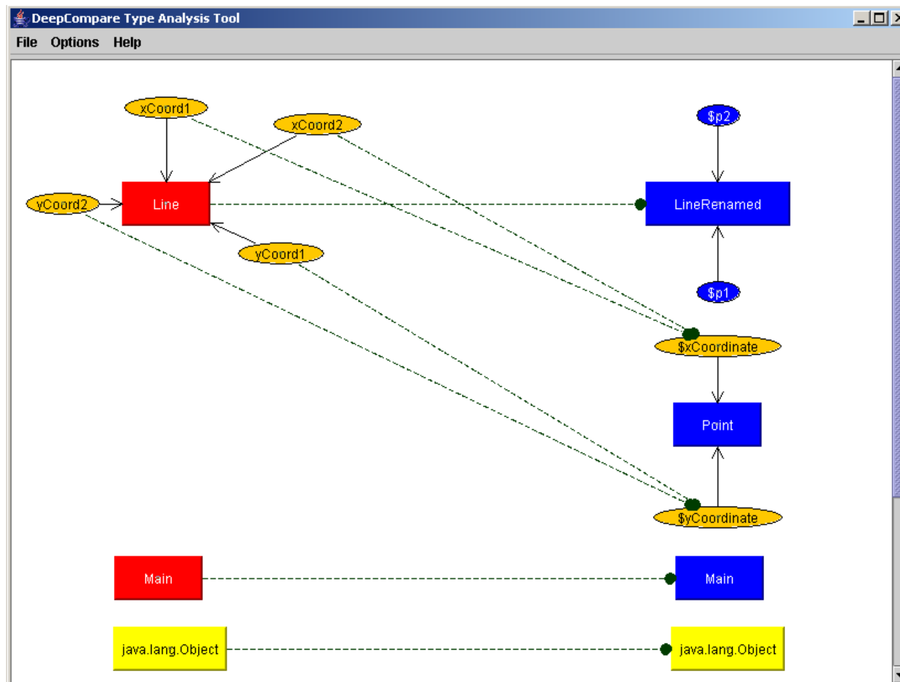


Figure 8: DeepCompare at work on the academic example. Similarity links between types of the original (left) and new version (right) are shown. Additional detected similarities between JDK types are not shown for clarity.

is between 65 and 85 classes. This roughly corresponds with what we consider to be the size of a large component. As such, JTeg clearly meets all requirements, except for the last, since it is not a component but an application.

However, this is not nearly as problematic as it may seem at first. We mentioned earlier that we base our component definition on the definition by Szyperski, but assume that the component is implemented as a collection of tightly coupled Java-objects. In this regard, any Java-application can be seen as a component: an application is a unit of deployment, and all communication with other components is explicit and asynchronous. Therefore, although a stand-alone application may not fit the intuitive image of a component, they are relatively similar. Only one significant difference remains: components have a central point of entry: their component interface. The implementation of this interface therefore constitutes a single point of entry to the entire object-tree of the component, whereas in an application, generally more than one rootnode can be found in the object-tree. In addition, JTeg is a multi-threaded application, while components are considered to be reactive entities (threads are controlled by the runtime middleware environment ([35]). However, while this clearly has impact on the runtime aspects of our methodology, none of these differences are an issue for the similarity comparison process that we wish to test.

Three different scenarios have been worked out. First, DEEPCOMPARE is used to analyze the difference between JTeg 0.6 and 0.6.3. This is a relatively small update that consists of a few bug fixes and a small functional extension. Since the scenario changes very little of the architecture of the system, we would expect DEEPCOMPARE to perform well. The results of this setup are discussed in section 6.1. In the second scenario, version 0.6 is compared with version 0.5. This was a major upgrade in which both functional enhancements (e.g. multiple language support, additional protocol support, ...) and architectural changes were implemented. Results are shown in section 6.2. For the third scenario (which is discussed in section 6.3), a version was checked out from CVS, and a specific refactoring was executed.

6.1 Bugfix scenario (0.6 → 0.6.3)

In this scenario, version 0.6.3 was compared with version 0.6. The cumulative effect of three small updates (0.6 → 0.6.1 → 0.6.2 → 0.6.3) had limited impact on the architecture of the system. A bug in the communication code of the client was fixed (the implementation of a method had to be adjusted), and a small functional extension was added. Whereas version 0.6 only supported one mission *Conquer the world*, 0.6.3 allowed for arbitrary missions to be assigned to the player. Three classes were added for that reason.

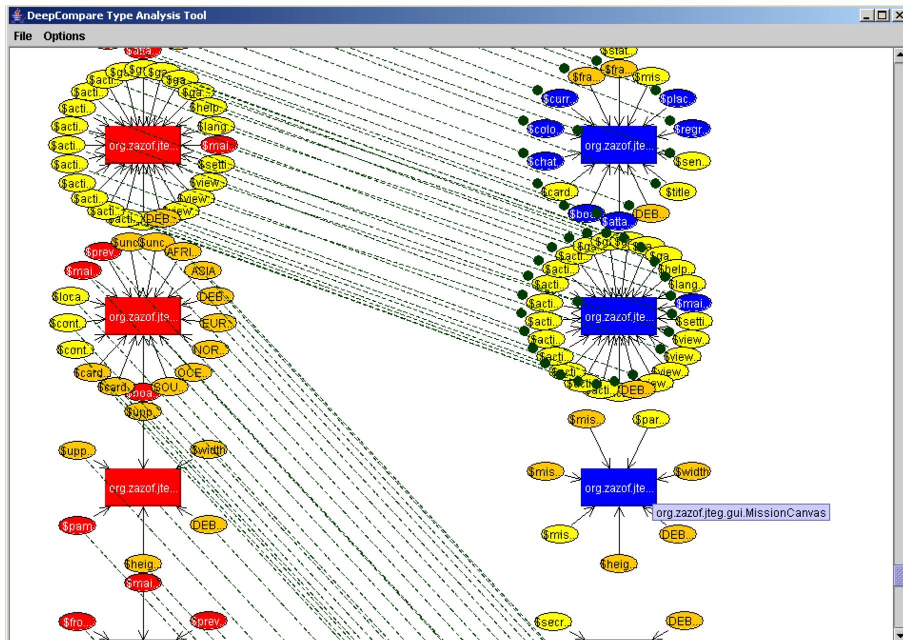
As expected, DEEPCOMPARE performed flawlessly. All 83 types that existed in version 0.6 were correctly matched with their counterpart in 0.6.3. The three remaining types were successfully marked as *added types*. In addition, 7 classes were directly changed to integrate the new functionality. 6 classes were indirectly modified. These results were manually verified and no errors were detected.

6.2 Major update scenario (0.5 → 0.6)

The second experiment required DEEPCOMPARE to compare two versions with significant differences. Between versions 0.5 and 0.6, JTeg underwent a series of major changes:

Feature enhancements: Multiple network protocols were added to the system, the menu-bar and the toolbar were implemented and multiple GUI enhancements were carried out.

Major refactoring: The internal workings of JTeg were rewritten: a state-machine was added to maintain consistency be-



(a) While the visualization is unpractical for a detailed analysis, its excellent for spotting trends. Links that do not run parallel with the others are candidates for manual verification. The addition of a new type is clearly visible.


```
<DeepCompareAnalysis>
 <Description>
   < TypesFirstProject  totalnumber="89" unmatched="0" matched="89"/>
   <TypesSecondProject totalnumber="89" unmatched="0" matched="89"/>
   <Changes direct="14"  indirect ="8"  influenced ="6">
     ...
   </Changes>
 </Description>
 <RemovedTypes/>
 <AddedTypes/>
 < IdenticalClasses >
   ...
 </ IdenticalClasses >
 <SimilarClasses>
   <SimilarClass  first ="org.zazof . jteg .MessageChainBuilder"
                  second="org.zazof . jteg .MessageChainBuilder"
                  changetype="Direct"  parental =" false ">
   <ImplementationImpact/>
   <StateImpact>
     <AddedMembers>
        <member name="$entryPoint">
           <link  otherparty ="$entryPoint_in_org . zazof . jteg . ProtocolFiveBuilder "
             linktype ="dc.comparison. harvester . PulledUpClassVariableHarvester "/>
        </member>
     </AddedMembers>
   </StateImpact>
   </SimilarClass>
    ...
   </SimilarClasses >
</DeepCompareAnalysis>
```


(b) Small Excerpt from a large and detailed XML file generated by DEEPCOMPARE(Portions of the XML file are not shown. The removed portions are marked by . . .). The XML notation is more detailed and better suited for manual verification.

Figure 9: Different representations of DEEPCOMPAREs results. Above, a graphical representation is shown of the bugfix scenario. Below, an XML representation is given from the Refactoring Scenario.

tween different GUI elements and all user-interaction was processed by a languagemanager to allow on-the-fly language changes. In addition, nearly all variables were renamed, since we decided to respect the Java naming convention for version 0.6.

While the tool did not find all correspondences without mistake, impressive results were achieved nevertheless which are summarized in table 1:

The significant feature enhancement resulted in the addition of 19 types, while no types were removed. DEEPCOMPARE identified the addition of 21 types, and the removal of 2 types. These two types were actually changed between version 0.5 and 0.6 but were not detected as such (first line in table 1). The first error was made for the `Message` type, which implements communication packets with the game server. While this was implemented as an interface in version 0.5, it was changed into an abstract class. Not surprisingly, this similarity was not detected. After all, with the exception of their name, very little resemblance between the two types can be found: they are situated in a different location in the inheritance hierarchy (a class extends `java.lang.Object`, an interface does not). In addition, both are small types: the original interface only consisted of a single method (`get_message_name()`), which was also renamed.

The second mistake made by our tool in this scenario was regarding a subclass of `Message`, which in version 0.5 had only dummy-functionality. Since almost nothing of its implementation corresponded and since it had no similar location in the inheritance hierarchy it was also considered to be an unrelated type. These two cases illustrate that DEEPCOMPARE does not exclusively depend on the naming of at type in its analysis. One interesting additional remark is that JTeg contains over 10 classes that implement the `Message` interface. All other classes were correctly identified since the similarity of their implementation was sufficient. It is clear that the incorrect detection of `Message` can easily corrected be modifying the chain and including a harvester that specifically searches for evolution from an interface to an abstract type and vice versa.

Evidently, identifying corresponding types is not sufficient. State transition requires an accurate mapping *within* those types as well. The performance on this area is reflected in the two last lines of table 1. A majority of the changes involved the renaming of variables and methods so that they corresponded to Java-style. Both for methods and variables, an accuracy of over 95% was achieved. Next to trivial renamings (eg `attack_armies` to `attackArmies`), a variety of less evident changes were detected (e.g. `place_armies` → `$locationsPlaced` and `get_location_y` → `getAbsoluteYCoordinate`). After manual verification of the results generated by DEEPCOMPARE, we found that no false positives occurred for member variables. In some cases, no match was found when a variable was indeed renamed. In all cases this was because either the type of the variable was also changed, or the name was so drastically changed that no correspondence could be found whatsoever.

False positives were found, however, during the identification of similar methods between two versions of a type. This is a result from the initial construction phase of our similaritymodel (see section 5.1). Since currently, none of the harvesters base their decisions on method-similarities, these false positives had zero impact on type matching.

6.3 Refactoring scenario (two untagged versions)

The last scenario consisted of a refactoring test. Using the well known open source IDE Eclipse, two refactorings were performed on JTeg version 0.6.3: the class `Country` was renamed to `Land`, and a variable was pulled up in the class hierarchy from `ProtocolFiveBuilder` to `MessageChainBuilder`. Since Eclipse offered to rename all variables with the character sequence "country" in their name to their counterparts with "land", we gracefully excepted this invitation, as we believe it to be realistic for a developer to agree with such suggestion. The main benefit of this third test-scenario is that, unlike the previous two refactorings, all changes are known with 100% certainty. We therefore know exactly what was changed, and what the tool should return. This is slightly more reliable than manual verification.

Although the tool perfectly identified both changes the class-renaming refactoring did locate a weakness of our current implementation. While logically only three types have really changed, DEEPCOMPARE identified 14 types in the *direct-change* category. The reason is that all methods whose implementation uses the `Country` type, or that contain variables of this type are said to have changed, even though at runtime the name of a type is not extremely relevant. It is important to note that these are not false positives. According to the definition, all 14 types were in effect directly changed. These results, albeit a little conservative, are correct and would lead to a correct replacement strategy. It is clear however, that further optimizations to the analysis are possible that would reduce the number of objects that are targeted for replacement when such a refactoring is executed. The search for these optimizations is future work.

6.4 Runtime performance of DeepCompare

DEEPCOMPARE performs its analysis offline. As such, runtime performance is not a key issue. Nevertheless, in order to be of any practical use, it is important that an analysis completes within a reasonable time-period. Our experience with the tool shows that this is clearly the case. Quantitative statistics on the performance of our tool are shown in table 2. All tests were carried out on a Pentium III 1Ghz desktop PC with 512mb of RAM running a Sun JDK 1.4.2 virtual machine on Windows 2000.

To ensure an objective and accurate estimate of the runtime performance, 4 different test cases were carried out which are shown as the rows of table 2. For each of the tests, measurements were taken during the three different stages of the analysis phase carried out by DEEPCOMPARE:

1. Timing of the source code parsing step and construction of the meta-model (column 1). This also includes additional type resolving and information lookup from the JDK base and library classes that are used by the component versions.
2. The construction of the similarity model (column 2). During this phase we measured the number of type-comparisons that were executed and placed in the similarity model. We also timed this entire process.
3. The duration of the harvester-chain analysis on the similarity model (column 3).

	Correctly Identified	Not Detected	False Positive
Types	52	2	0
Methods	148	0	6
Member Variables	111	5	0

Table 1: Results from the comparison between $0.5 \rightarrow 0.6$

	Meta-Model	Similarity Model	Harvester Chain
NanoXML (21 types) vs. LiteratureProcessor (7 types)	7 seconds	230 type comparisons in 2 seconds	4 seconds
JTeg 0.5 (64 types) vs. JTeg 0.6 (84 types)	13 seconds	5772 type comparisons in 55 seconds	41 seconds
JTeg 0.6 (84 types) vs. itself	16 seconds	7417 type comparisons in 94 seconds	37 seconds
JTeg 0.6 (84 types) vs. JGraph (76 types)	20 seconds	6701 type comparisons in 67 seconds	680 seconds

Table 2: Performance characteristics of DEEPCOMPARE.

Each of these three stages are shown as the columns of table 2. The first test-setup was the comparison of two very small, but radically different applications: an XML parser (NanoXML: 21 classes) and a home-brewed tool to sort electronic literature (7 classes). Due to its small size, the results from this analysis are mainly important as a base for comparison. The second line of table 2 are measurements taken from the scenario that was discussed in section 6.2. The two final test are chosen to investigate the impact of similarity between the two version on the runtime performance of our tool: we compared JTeg 0.6 to itself (perfect similarity) and to an entirely different program (JGraph, a package to visualise graphs in Java).

As seen in the second column of table 2 the time required to construct a similarity model is directly proportional to the number of type comparisons: $O(n \times m)$, with n and m the number of types in the first and second version respectively[6]. Each such comparison requires on average 10ms of computation on our test setup. There is a large variance on this number, since the complexity of a type-comparison depends on the number of methods, members and interfaces. However, due to the large number of these comparison, simple and complex type comparisons cancel out resulting in a relatively stable average.

Once the similarity model is constructed, the harvester-chain is executed. Computational complexity strongly depends on the similarity of the two projects. Strong similarity between the projects drastically reduces the duration of the analysis. This is because identical types and members can be easily detected and are eliminated by the first harvesters in our chain. As such they are exempt from further (more computationally complex) analysis. If a component is compared to itself, the harvesterchain will terminate very quickly whereas a comparison with an entirely unrelated piece of software uses more expensive steps to find similarities. This is clearly illustrated in the last two lines of table 2. Although JTeg 0.6 has slightly more classes than JGraph, and while its types are on average more complex (resulting in a slower construction of the similarity model), the execution of the harvesters is 18 times faster than in the scenario where no similarity is present between the two components. Comparing unrelated pieces of software is not meaningful in practice however, and typical evolution scenarios tend to have short computation times.

While this test-setup is not proof of efficiency, performance of DEEPCOMPARE is clearly not an obstacle. We also would like to stress that no attempt has been made to optimize the runtime performance of our tool. It is very likely that speed can be increased and the memory consumption[7] can be reduced if the necessary profiling and optimizations are performed on our codebase.

[6]Note that n and m may also include used types from the JDK or external libraries.

[7]Currently, memory consumption is rather high, about 150MB for a complex project – mainly caused by the meta-model instance.

7. RELATED WORK

The problem of live updates is not new. Since it was introduced by Fabry in 1976 [10], a substantial amount of work has been done in this domain which has resulted in a variety of different systems. However, nearly all the past-research in this field exclusively focuses on the dynamic aspects. State transfer is either ignored completely, or it is left up to the programmer to manually implement a transition function. As such, our work is orthogonal to most work done in the field and the ideas of this paper can complement existing systems. We begin this section with an overview of related work in the field of runtime evolution for procedural, object oriented and component oriented systems. We also discuss to what extend the systems under discussion offer support for state transfer. The second part of this section presents related work on static analysis and type conversion in persistent storage and databases.

Solutions for procedural systems are discussed in [12, 15, 17]). Using some form of indirection, a method-call is diverted to a new version of the procedure. Wrappers are used to handle interface changes and in some cases state is transferred using mapper functions. More recently, Hicks described a system for updating `Popcorn` (a type safe C-like language). The system uses dynamic patches and allows for the update of code, data and types at a moment chosen by the application programmer ([17]). The patches consist of verifiable native code and contain both the code of the new version and the transition code to be used during the update. Although some tool support is included to construct these patches, it is limited to the generation of a framework in which the user can implement the transition. Very little intelligent support is offered for the transfer of the state itself.

Other systems focus specifically on dynamic Java. In [22], Malabarba et al. focus on being fully consistent with the Java language and its type safety. They extended the default Java class loader in a way that class definitions can be replaced and objects or dependent classes can be updated. The replacement is initiated by the user by explicit calls to the class loader in the application program. With Gilgul, Costanza proposed an extension to the Java language that supports Dynamic Object Replacement [4]. In order to achieve this, Gilgul introduces some indirections. Nevertheless,

the overhead is never very large, and occurs only during object replacements. An implementation of Gilgul is available for the Kaffe Virtual Machine. Techniques used in the implementation of both DVM and Gilgul can be used to implement the pointer relaying for indirectly changed types.

With Conus, Kramer and Magee pioneered in the field of component-oriented evolution. In [19] they describe a system in which a programmer can specify the desired changes in a declarative way. They introduced the concept of quiescent nodes for safe removal of a component from the running system. They introduced algorithms that can be used to determine that a component is put in a safe state before it is replaced, even when it participates in transaction. Their work is mainly concerned with change management and does not focus on the replacement of the component itself. In [16], Hauptmann and Wasel describe a system for on-the-fly updating an embedded system based on provisions available in the Chorus operating system. Software applications are made replaceable in a separate step during development. An important aspect of their work is that deterministic timing behaviour is achieved under certain conditions.

A very flexible approach to software evolution can be found in meta-architectures (e.g. [8, 25]). Through reification of object-oriented concepts (class, method-call, ...), a meta-model is built on top of the application. By changing this meta-model, structural changes of the application are possible. A major advantage is that this provides a clean separation between the application code (base-code) and the code responsible for the reconfiguration (meta-code). In some cases[8] it is possible to build an adaptation framework that is orthogonal to the application domain [9]. Since their models are runtime entities that need to be consistent with the application state, their technique results in a large overhead. However, the technique offers possibilities to conduct off-line analysis of components to generate state-transfer functionality. While this is not implemented, DEEPCOMPARE can in theory be coupled with their runtime meta-model to compare a running component with a new version.

An interesting approach to implement dynamic reconfiguration is Kniesel's work on *dynamic delegation* with Lava (a variation of Java) [14]. It allows for interface changes and does not require the application to be developed with evolution in mind. The concept of delegation is very well suited for component-oriented systems, since it allows child references to its parent to change dynamically. For delegation based approaches, our work is not relevant since no state is ever transferred: a component is never replaced, but wrapped with new functionality. However, delegation will retain all previous versions of a component in memory, leading to a significant memory overhead. In addition, the redirection of `self`-calls with delegation must be strictly disciplined to prevent corruption of object integrity. Problems with delegation are discussed in detail by Truyen ([30]) and Steyaert ([28]).

To give an exhaustive overview of existing systems for runtime evolution is beyond the scope of this paper, and we refer to [17, 32] for more complete surveys.

Considering static analysis techniques, the Eclipse-based Chianti tool by Ren et al. ([26]) performs an analysis which is similar to DEEPCOMPARE's change construction stage (discussed in section 5.1) . Chianti analyzes two versions of an application and decomposes their differences into a set of atomic changes. The purpose of their analysis is very different however, since their goal is to determine affected regression tests. In [27], Robillard and Murphy describe a tool named FEAT which can capture different parts of the implementation of a concern in order to increase code comprehension and to ease its manipulation. Similar to DEEPCOMPARE, static analysis of Java source code is used in order to extract semantic meaning and correlations.

[8]Some patterns of adaptation might require application specific knowledge. Therefore, not all adaptation are possible while retaining orthogonality with the application domain.

Also related to our work is evolution of persistent storage. In [6], DMitriev introduced the PJama Persistent Language in which the default Java VM was extended with an API to transparently add persistence to applications. The persistent store can then be used to achieve evolution offline. Two techniques are offered: *default conversion* and *conversion*. Although the first type automatically executes a state transfer, little intelligent mapping between types exists. Only fields whose name and type are identical in both versions are copied. All other fields are initialized with default values. The latter technique is more flexible, but requires the programmer to manually implement the transfer. In [21, 20], Staudt-Lerner exploits the structure of types to convert data stored in databases after the underlying database-scheme was changed. Derivation rules specify how a type can be derived from one or more other types. A few patterns (such as type inlining or encapsulation) that often occur when executing scheme changes are presented and are taken into account by her algorithms.

One last research-track that deals specifically with the problem of deriving correspondence between different structures, is the theory behind plagiarism detectors ([38, 23, 36]). Plagiarism detectors attempt to extract similar programs from a large population. Although many of the techniques used by plagiarism detectors are probabilistic in nature (i.e.: they only work well with larger code-segments) and use pure string comparison techniques, some of the techniques can be used in the implementation of our methodology. For instance, the underlying mechanisms on which DEEPCOMPAREs implementation of similarity detection is based, is derived from techniques presented in [23, 37].

8. LIMITATIONS AND FUTURE WORK

While the results we presented in this paper are promising, our approach has some limitations. The very nature of static analysis requires that type information can be derived from the source code at design time. While purely dynamically typed languages such as Smalltalk do offer other advantages for dynamic evolution (eg Smalltalk's extremely powerful reflection mechanism makes it excellent for dynamic evolution), attempts to find similarities between different versions would have to be based solely on naming and location, resulting in much less reliable and accurate results. Next to typing information, our analysis is also strongly tied to the object-oriented nature of the internals of a component. Although DEEPCOMPARE is designed specifically for Java-based component-implementations, it can be easily modified to support other input that can be mapped to our underlying meta-model such as C# or bytecode, simply by implementing a different syntax-parser.

In order to further improve the recognition of equivalent structures, research will be done on the implementation of more complex harvesters. Since refactoring more and more dominates the evolution of software, the refactorings described in [11] will be used as a starting point to add more harvesters to the tool.

It is an intrinsic property that accurately detecting correspondence is harder if the changes are more extensive. We will therefore attempt to couple DEEPCOMPARE to a version control system as CVS or subversion. DEEPCOMPARE can then exploit information from intermediate versions inside the version repository to further increase its accuracy.

A topic left untreated by this paper is correctness. Nevertheless, by its very nature, dynamic software evolution requires that all steps in the process execute without error since failure could result in an incorrect program state or system crash. One approach would be to formally proof correctness of a given state transfer. However, this is extremely hard to do and is likely to be very specific for a given program and therefore limited in its applicability. Therefore, we will extend DEEPCOMPARE with the capability to flag possibly problematic structures to the user (e.g. structures that would take too long to convert during a *live* update, complex conversions that require manual verification, ...). In addition, the runtime environment will be extended with a state invariant checker that verifies whether the transferred state complies with the invariants specified by the programmer, and initiate a rollback if this is not the case.

Finally, future work also includes a full implementation of the code generator and its type transformers, and the integration with a preprocessor that was already developed for our middelware platform. The code generator is the final step in completing the entire tool chain that supports our methodology as shown in figure 1, from static analysis (DEEPCOMPARE) all the way up to the runtime replacement of the component on our component middleware platform, DRACO ([35]).

9. CONCLUSIONS

In this paper we introduced our methodology to component-based evolution, and our approach to state transfer in particular. At design time, components are instrumented with functionality to transfer their state during the replacement. The developer is aided in this task by our tool: DEEPCOMPARE. This tool conducts an analysis of the (object-oriented) source code of different versions of a component. A classification is presented which introduces different categories of changes, according to the impact of the change on the set of objects that make up the component: *direct*, *indirect* or *influenced*. In addition, different heuristics are described that are used to identify equivalent data structures between different versions. Implemented in the form of harvesters, these algorithms are placed in a chain and can be easily customized by the user to increase the accuracy of our tool. The information generated by DEEPCOMPARE is used to (partially) generate the state transition functions that will executed during the component replacement at runtime. In the second part of our paper, our methodology and its implementation are evaluated, first on an academic example, then on a larger project: a Java-based game. Three scenarios show that, even with only 5 harvesters, accuracy is very high: over 95% of all state variables were correctly mapped to their counterpart in the new version.

10. REFERENCES

[1] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Language support for connector abstractions. *Lecture Notes in Computer Science: ECOOP 2003 - Object Oriented Programming*, 2743:74–102, 2003.

[2] K. Beck. *Extreme Programming Explained: Embrace Change*. 1999.

[3] Y. Berbers, P. Rigole, Y. Vandewoude, and S. V. Baelen. Components and contracts in software development for embedded systems. In L. De Backer, editor, *Proceedings of the first European Conference on the Use of Modern Information and Communication Technologies*, pages 219–226, 2004.

[4] P. Costanza. Dynamic object replacement and implementation-only classes. In *Proc. of Workshop on Component-Oriented Programming at ECOOP*, 2001.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proc. of OOPSLA*, pages 166–177, 2000.

[6] M. Dmitriev and M. Atkinson. Evolutionary data conversion in the pjama persistent language. In *Proceedings of the 1st ECOOP Workshop on Object-Oriented Databases*, Lisbon, Portugal, June 1999.

[7] J. Dockx, N. Smeets, K. Mertens, and E. Steegmans. jnome: A java meta-model in detail. Technical Report CW323, KULeuven Department of Computerscience, December 2001.

[8] J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In *Reflection*, pages 81–88, 2001.

[9] J. Dowling, T. Schaefer, V. Cahill, P. Haraszti, and B. Redmond. Using reflection to support dynamic adaptation of system software: A case study driven evaluation. In *OOPSLA Workshop on Object-Oriented Reflection and Software Engineering*, pages 169–188, Denver, Colorado, November 1999.

[10] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 470–476. IEEE Computer Society Press, 1976.

[11] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, 2000.

[12] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *The Journal of Systems Software*, 14(2):111–128, February 1991.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[14] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proc. of ECOOP 99*, Lisbon, Portugal, June 1999.

[15] D. Gupta. *On-line Software Version Change.* PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, Nobember 1994.

[16] S. Hauptmann and J. Wasel. On-line maintenance with on-the-fly software replacement. In *Proceedings of the 3th International Conference of Configurable Distributed Systems*. IEEE Computer Society Press, 1996.

[17] M. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, June 2001.

[18] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards transparent hot-swapping support for producer-consumer components. In *Second Int. Workshop on Unanticipated Software Evolution*, pages 9–16, Warshau, 2003.

[19] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306.

[20] B. S. Lerner. TESS: Automated support for the evolution of persistent types. In *Proc. of the 12th Automated Software Engineering Conference*, pages 172–182, November 1997.

[21] B. S. Lerner. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems*, 25(1):83–127, March 2000.

[22] S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J. F. Barnes. Runtime support for type-safe dynamic java classes. In *Proc. of ECOOP*, 2000.

[23] G. Malpohl, J. Hunt, and W. Tichy. Renaming detection. In *Proc. of the 15th International Conference on Automated Software Engineering*, 2000.

[24] A. Rausch. Software evolution in componentware - a practical approach. In *Proc. of the Australian Software Engineering Conference*, 2000.

[25] B. Redmond and V. Cahill. Iguana/j: Towards a dynamic and efficient reflective architecture for java. In *Workshop on Reflection and Meta-Level Architectures at 14th European Conference on Object-Oriented Programming*, June 2000.

[26] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 432–448, Vancouver, BC, Canada, October 26–28 2004.

[27] M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 406–416, 2002.

[28] P. Steyaert and W. D. Meuter. A marriage of class-and object-based inheritance without unwanted children. *Lecture Notes in Computer Science: ECOOP 1995 - Object Oriented Programming*, 952:127–144, 1995.

[29] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, November 2002.

[30] E. Truyen. *Dynamic and Context-Sensitive Composition in Distributed Systems*. PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, November 2004.

[31] D. Urting, S. V. Baelen, T. Holvoet, P. Rigole, Y. Vandewoude, and Y. Berbers. A tool for component based design of embedded software. In *Proceedings of Tools Pacific 2002*, Februari 2002.

[32] Y. Vandewoude and Y. Berbers. An overview and assessment of dynamic update methods for component-oriented embedded systems. In *Proc. of Software Engineering Research and Practice*, pages 521–527, June 2002.

[33] Y. Vandewoude and Y. Berbers. Supporting runtime evolution in SEESCOA. *Journal of Integrated Design & Process Science: Transactions of the SDPS*, 8(1):77–89, March 2004.

[34] Y. Vandewoude and Y. Berbers. Component state mapping for runtime evolution. In *Submitted to the Technical Session on Support for Unanticipated Software Evolution (USE 2005)*, Las Vegas, 2005.

[35] Y. Vandewoude, P. Rigole, D. Urting, and Y. Berbers. Draco : An adaptive runtime environment for components. Technical Report CW372, Department of Computer Science, Katholieke Universiteit Leuven, Belgium, December 2003.

[36] K. Verco and M. Wise. Software for detecting suspected plagiarism: Comparing structure and attribute-counting systems. In *Proceedings of the 1st Australian Conference on Computer Science Education*, Sydney, Australia, July 1996.

[37] M. J. Wise. String similarity via greedy string tiling and running karp-rabin matching. `http://www.bio.cam.ac.uk/~mw263/ftp/doc/RKR_GST.ps`, 1993.

[38] M. J. Wise. Improved detection of similarities in computer program and other texts. In *Twenty-Seventh SIGCSE Technical Symposium*, Philadelphia, 1996.