Universiteit Antwerpen

Vrije Universiteit Brussel

The Implementation of the CHA-Q Meta-Model
A Comprehensive, Change-Centric Software Representation

Cha-Q project deliverable 2.1b

ANSYMO (Universiteit Antwerpen) — SOFT (Vrije Universiteit Brussel)

Responsible Christophe Scholliers (VUB)
Authors Christophe Scholliers (VUB), Coen De Roover (VUB),
Alessandro Murgia (UA), Javier Pérez (UA)

Contents

1 Introduction **7**

2 Overview of the CHA-Q Meta-Model **9**

3 Implementation Highlights **11**
3.1 Property Annotations 11
3.2 Memory-efficient State Tracking 12
3.3 Reflection Cache 13
3.4 Persistence Considerations 13

4 Persistence **15**
4.1 Graph Representation 15
4.2 Example Serialization: Method Declaration 16

5 Modeling Entities in the CHA-Q Meta-Model **19**
5.1 Java AST Nodes 19
5.2 Applying Changes 19

6 Evaluation **23**
6.1 Test Data 23
6.2 Storage and Memory Performance 24

7 Conclusion **25**

8 References **27**

List of Figures

2.1 Overview of key CHA-Q meta-model elements. 10

3.1 Cha-Q annotations 12

4.1 CHA-QGraph Database Model 16
4.2 Graph View of a Stored Method Declaration 16

5.1 Variable declaration ASTNode 20
5.2 Annotations in the VariableDeclaration class. 20
5.3 Creating and adding entity states to a snapshot. 21
5.4 Applying changes over the entity states. 21

6.1 Storage overhead of the CHA-Q meta-model versus a naive implementation. 24

List of Tables

1 Introduction

The CHA-Q project (Change-centric Quality Assurance)[1] aims to strike a balance between agility and reliability through change-centric quality assurance tools. This document reports on CHA-Q project deliverable 2.1b: the implementation of the CHA-Q meta model, we give a detailed overview of its object-oriented API, the persistency through a graph database, and a strategy for tracking the history of artefacts in a memory-efficient manner.

Within the software engineering community, the use of meta-models to provide common representation frameworks that can be leveraged by various software engineering tools is not new. One example of such a generic meta-model to represent object-oriented systems is the FAMIX meta-model [1]. FAMIX offers a language-independent, first-class representation of object-oriented, class-based languages that has been used by a wide range of software engineering tools such as the MOOSE reverse engineering tool suite. FAMIX3 [2] represents the most recent incarnation of this meta-model.

Next to such representations of object-oriented programs, a body of work exists with regard to modeling multiple versions of a system. A good overview of the early research in this area can be found in the book chapter by D'Ambros et. al. [3]. HISMO [4] extends the FAMIX meta-model such that multiple versions of a software system can be represented. For each version in the history of the system, a complete model of that version — along with information that relates source-code entities over various versions — is stored. The more recent Orion [5] meta-model also represents multiple versions of FAMIX entities, but it does this in a manner that avoids copying entities that have not changed between versions. Its strategy has not only been observed to result in memory-efficient models, but these models can also be constructed faster as fewer allocations have to be performed.

Another body of work relies on meta-models that represent change operations to a system as first-class objects. The SpyWare tool suite by Robbes et al. [6, 7], in contrast, records all changes that are made to a system using the integrated development environment (IDE). Internally, SpyWare provides a fine-grained model where each individual change to the system is stored. The ChEOPS [8] system by Ebraert et al. offers a similar meta-model for representing and storing changes. Both approaches target Smalltalk. Another similar approach for the reification of changes is the one taken by Hattori [9] in Syde, a tool that logs the changes made by several developers in parallel. Syde targets Java. The UniCase [10, 11] tool represents changes to EMF models as first-class entities to facilitate conflict detection

[1] http://soft.vub.ac.be/chaq/

and resolution. The tool suite around OperationRecorder [12] employs an extremely fine-grained representation of edit operations to text.

So far, we have only discussed meta-models that represent the state, history or individual changes to the source code of a system. The most influential meta-model for representing issue tracking information is the one used by the Evolizer [13] tool suite. An earlier version is detailed along with a representative meta-model for versioning meta-data (e.g., commit messages) in [14].

While existing meta-models define a representation of source code or changes, none provides a complete representation of both —let alone of the other artefacts of a software system (versions, issues, mailing lists, . . .) and their changes.

In this paper, we introduce the CHA-Q meta-model that owes its name to the CHA-Q project (Change-centric Quality Assurance)[2]. This project aims to strike a balance between agility and reliability through change-centric quality assurance tools. These tools are to share a first-class representation of the artefacts that comprise a software system (source code, files, bugs, bug comments, mailing lists, . . .), as well as the complete history of all individual changes to these artefacts —a representation defined by the CHA-Q meta-model. The main contributions of this new meta-model are:

- A first-class representation for changes to various software artefacts.
- A uniform and extensible object-oriented API.
- An implementation that uses a graph database for persistency, while tracking the history of software artefacts in an memory-efficient manner.

The latter relieves developers from secondary concerns such as memory usage and storage requirements. In the remainder of this paper we give an overview of our meta-model chapter 2, highlight important properties of its implementation in chapter 3 and the underlying persistence strategy in chapter 4. Subsequently, we demonstrate the meta-model in chapter 5 by creating models for a use case, and present the results of a preliminary performance evaluation in chapter 6.

[2] http://soft.vub.ac.be/chaq/

2 Overview of the CHA-Q Meta-Model

The CHA-Q meta-model defines a representation of the various artefacts that comprise a software system, as well as the complete history of all individual changes to these artefacts. Based on our experiences with the FAMIX [1], ChEOPS [8] and Ring [15] meta-models, we have opted for an object-oriented representation. Figure 2.1 depicts its high-level UML class diagram.

Changes are modeled as *first-class* objects that can be analyzed, repeated and reverted (cf. `Change`). To this end, we provide a representation of the dependencies between two changes (cf. `Change-Dependency`). These imply a partial ordering within a given set of changes (cf. `ChangeSet`). The corresponding elements are depicted in **blue**. Similar meta-models have already proven themselves for representing changes to code (e.g., SpyWare [7], ChEOPS [8] and Syde [9]) and to EMF models (e.g., UniCase [10]). Our meta-model goes beyond the state of the art by representing changes to the properties of any system artefact (i.e., source code, files, commits, bugs, e-mails, ...) in a uniform manner. This uniform treatment of an artefact's properties is inspired by the reflective API of the Eclipse JDT. The corresponding elements (cf. `PropertyDescriptor`) are depicted in **brown**.

Applying a change results in a new state for its subject (cf. `EntityState`). Figure 2.1 depicts the corresponding elements in **yellow**. Examples include abstract syntax trees (cf. `ASTNode`) and issues managed by an issue tracker (cf. `Issue`). The meta-model elements related to issue tracking and e-mail communication are inspired by the meta-model used by the Evolizer [13] and STNACockpit [16] tools respectively. Figure 2.1 depicts them in **green**.

Snapshots correspond to the state of all of a system's artefacts at a particular point in time as seen by a particular developer (cf. `Snapshot`). The delta between two snapshots is a set of changes (cf. `ChangeSet`). Snapshots of the entire system can be inspected and compared. This connection is similar to the one between Ring's history and change meta-model [15]. Revisions (cf. `Revision`) are snapshots placed under control of a version control system. Figure 2.1 depicts the corresponding elements, such as a modification reports and branches, in **pink**. These are inspired by the revision meta-model used by Evolizer [13].

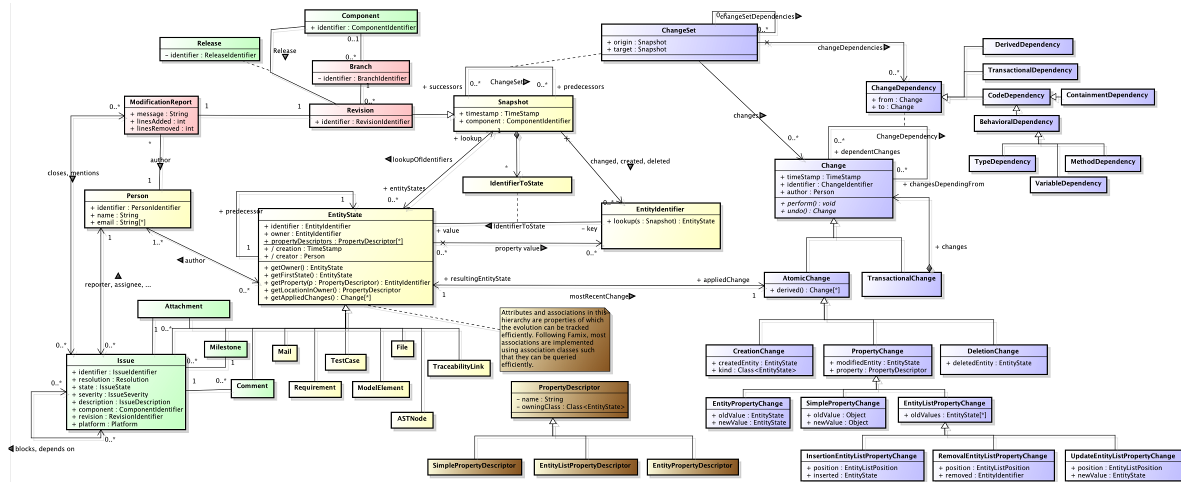

Figure 2.1: Overview of key CHA-Q meta-model elements.

3

Implementation Highlights

The CHA-Q meta-model associates a unique identifier (cf. `EntityIdentifier`) with each change subject (cf. `EntityState`). This enables tracking the evolution of a single subject throughout the history of a system. For each subject, a history of previous states is kept in a memory-efficient manner; successive states share the values of properties that do not change. We deem this necessary as copying of entity states has been observed to consume 3GB of memory for the Syde change-centric representation of a version repository of 78MB [9]. However, a selective cloning approach would be impractical to implement as all entities are interconnected transitively. We therefore follow the approach advocated by the Orion [5] and Ring [15] history meta-models. Property values are identifiers (cf. `EntityIdentifier`) that are looked up with respect to a particular snapshot.

To ensure that this additional level of indirection does not endanger type safety, our implementation relies on Java generics and property annotations. The property `initializer` of a `VariableDeclaration`, for instance, can only have `Expression` identifiers as its value. As shown in chapter 5, the programmer can easily enforce such constraints in the CHA-Q meta-model.

Despite this memory-efficient representation, the working memory of a typical development terminal is unlikely to suffice for the entire history of the industry-sized projects that we aim to support. Our implementation therefore persists instances of meta-model elements to a Neo4J[1] graph database and retrieves them on a strict as-needed basis. Use of weak references ensures that instances that are no longer needed can be reclaimed by the garbage collector. Our two-way mapping is driven by run-time reflection about the aforementioned property annotations. This renders our implementation extensible. Extensive caching ensures that reflection does not come at the cost of a performance penalty.

3.1 Property Annotations

The CHA-Q meta-model ensures type safety of the fields of the `EntityState` by means of annotations. As shown in figure 4.2, there are three annotations defined in the `be.ac.chaq.model.entity` package.

- **SimpleProperty**: This annotation can adorn the fields of any class that extends

[1] http://www.neo4j.org

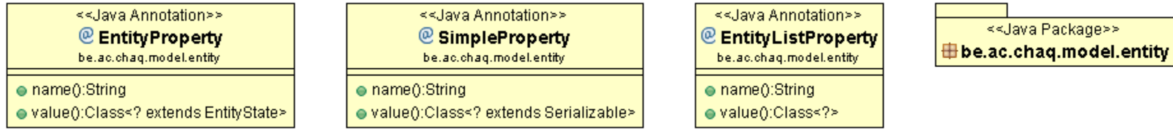

Figure 3.1: Cha-Q annotations

EntityState. All fields annotated with the SimpleProperty annotation must be Serializable. When a field is annotated with a SimpleProperty access to this field is provided through the getProperty method provided by the EntityState class. Note that adding the annotation to his class is the only thing the programmer has to do.

- **EntityProperty**: This annotation is used to flag that the annotated field is used in order to store another EntityState object. Again this property can be read by making use of the getProperty method defined in the EntityState class.

- **EntityListProperty**: This annotations is very similar to the EntityProperty annotation but in stead of indicating a single field that refers to an EntityState this annotation must be used to indicate a list of entity states.

In chapter 5 we give an overview of how the meta-programmer uses these annotations in order to create a custom EntityState.

3.2 Memory-efficient State Tracking

As mentioned before, copying an entity each time its state changes (the approach taken by Syde [9]) would be prohibitively expensive for large projects. To minimize memory consumption, successive EntityState instances have to share the values of properties that do not change. As all entities are interconnected transitively and these connections can be navigated in multiple directions (e.g., from a method to its declaring class and from a class to its declared methods), consistency would be difficult to maintain using a selective shallow and deep cloning approach.

Following Orion [5] and Ring [15], we store the values of properties as EntityIdentifier instances that have to be looked up starting in a particular snapshot. Instances of class Snapshot correspond to the state of all of a system's artefacts at a particular point in time as seen by a particular developer. To this end, each snapshot maintains IdentifierToState mappings from the unique identifier of an entity to its current state within the snapshot. Performing a PropertyChange therefore amounts to making a shallow clone of the current EntityState of the change subject, updating the Snapshot's current mapping from the subject's EntityIdentifier to the new EntityState of the change subject, and updating the property's value in the newly created EntityState. Deleting an entity amounts to removing its IdentifierToState mapping, rendering the corresponding entity inaccessible in the current snapshot. Each snapshot does keep track of all changed, created and deleted entity identifiers for inspection purposes.

Increased access cost is the price to pay for this sharing of property values that do not change between successive states of an entity. Implementation-wise, indirect lookups can

by hidden by having accessor methods return proxies that wrap an entity identifier with a snapshot and forward all requests to the corresponding entity state. Orion [5] and Ring [15] rely on a similar proxy to provide snapshot-unaware tools a view on a particular snapshot.

Note that our meta-model's snapshots comprise a middle ground between a complete version-based and a complete change-based representation of a system's evolution. Depending on their timespan, snapshots can accumulate the effect of a single or of several changes. As such, they can be used to represent eras of a system's lifetime about which fine-grained change information is unavailable or not desired.

3.3 Reflection Cache

The annotations defined by the programmer are evaluated at runtime by making use of reflection. These reflection operations are relatively slow and therefore, the implementation of the meta-model makes use of a cache to avoid having to repeat the reflective operations over and over again.

In the implementation of the `EntityState` class getting and setting properties are done by making use of a `PropertyDescriptorsMap`. For every class this map contains a property descriptor for each of the fields of the entity state. The programmer does not need to built this map himself. Instead, whenever a property of a class is requested the property descriptor map is consulted to retrieve the map for this particular class. If there is no entry in the property descriptor map a new entry is built by using reflection. The constructed property descriptor map is then stored in the property descriptors map for later retrieval. Because this map is cashed in the entity state class the reflective operations are kept to a minimum.

3.4 Persistence Considerations

While the CHA-Q meta-model already provides a lot of beneficial measurements in order to make efficient use of memory, loading all revision of a large software artefact in memory is currently not possible. Therefore, the CHA-Q meta-model makes use of weak references to ensure that instances that are no longer needed can be reclaimed by the garbage collector. This does not mean that the garbage collected references are simply thrown away. The meta-model automatically persists the snapshots of the project that can not be fit into the memory and restores these snapshots when they are needed in order to perform an analysis. However, the programmer which makes use of the CHA-Q meta-model is not confronted with the way the meta-model is serialised or deserialised. Form the programmers point of view all the versions of the project can be consulted. In the next section we give a more in depth overview of the persistence model of the CHA-Q meta-model.

4 Persistence

We have opted to make use of a graph database in order to persist the CHA-Q meta-model. Instead of storing data in tables, a graph database represent the stored data by making use of nodes, edges and properties. The advantage of using a graph database instead of more conventional databases is that there is a better mapping of the object-oriented representation of the CHA-Q meta-model onto a graph than onto tables. For example, a node in the database that represents a method declaration simply has an outgoing edge to the body of that method just like in the object-oriented representation. Moreover, once a node in the graph database has been identified, retrieval of semantically close nodes can be done very fast as this is just a matter of following the outgoing edges.

4.1 Graph Representation

The size of the projects that we aim to support is much bigger than the memory size of a normal desktop computer. Therefore, the persistence of the meta-model is not only important for storage and later retrieval it is an essential part of the meta-model to deal with large projects. There is a one-to-one mapping between how the graph database stores nodes and relationships and the Cha-Q meta-model classes. Most nodes in the database represents an `EntityState` and have a set of links which point to entity identifiers nodes. Figure 4.1 shows the graph database model. At the top of the figure there are special nodes which map to snapshot in the CHA-Qmeta-model. These snapshots contain references to entity identifier nodes which in their turn have references to the actual entity nodes.

`EntityState` nodes have properties in order to identify from which class they were serialized in the database. EntityStates also have relationships to other nodes in the graph database. In particular, an entity node can have three kinds of relations to other nodes in the graph database. These type of references directly correspond to the type of properties an `EntityState` can have. The first kind of reference an `EntityState` node can have are simple entity state references. These references must point to simple entity state nodes which contain the state as one of their properties. Second, an `EntityState` node can have entity property references which point directly to other entity identifier nodes. These references must be followed in the graph database in order to get hold of the actual `EntityState` . Finally, the last type of reference is a entity list property reference which points to a special list node.

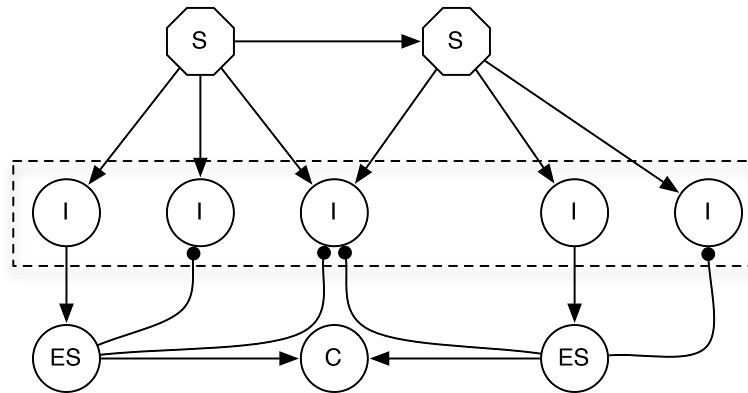

Figure 4.1: CHA-QGraph Database Model

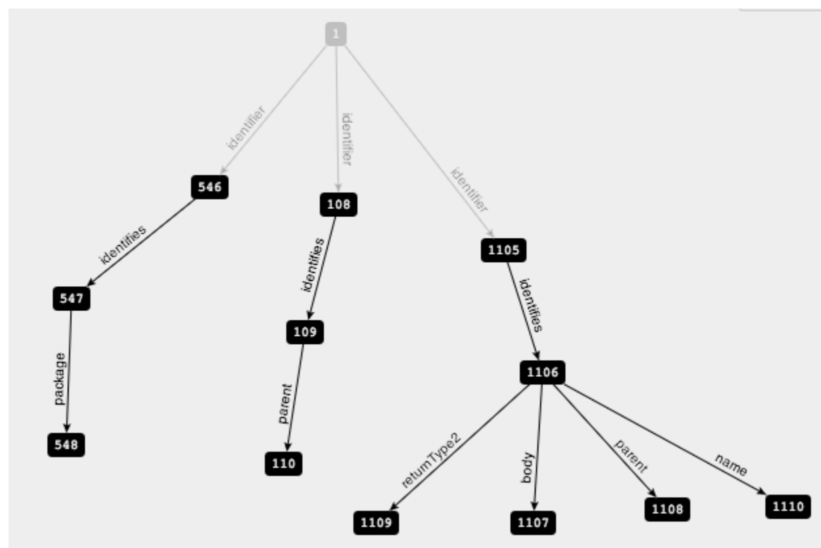

Figure 4.2: Graph View of a Stored Method Declaration

4.2 Example Serialization: Method Declaration

In this section, we give a graphical overview of the graph serialization process. As all the entity states of the software project are translated into nodes, showing even a medium sized project would be overwhelming. Therefore, we only show a very small fraction of a method declaration.

The serialization graph can be seen in Figure 4.2. Besides the actual method declaration we also show the node that represent the snapshot. As shown before a snapshot does not have direct references to the corresponding entity states. Instead a snapshot has a number of outgoing relationships which point to entity identifiers. In the example the snapshot node has number *1*. From all the outgoing edges only three are shown. In practice the snapshot has much more than three outgoing edges but in order to keep the overview they are not shown here. The leftmost edge points to a node with number *1106*, this node represents

the method declaration. As expected this node has a number of fields that are represented again with outgoing edges. In the case of a method declaration a node has a `ReturnType2`, a `body`, a `parent` and a `name`. Just as in the object model, the body declaration is not an `EntityState` but a `EntityStateIdentifier` that can be use in order to retrieve the `EntityState` from the snapshot.

5 Modeling Entities in the CHA-Q Meta-Model

In this section, we given an overview of how to use the meta-model by showing the modeling process for the definition of a custom `EntityState`. The entity state of our concrete use case is part of the Java AST-Node meta-model. We have implemented entity states for all Java AST-Nodes but here we limit ourselves to the implementation of a single AST-Node: `VariableDeclaration`. The implementation of the other Java AST-Nodes follows the same recipe.

5.1 Java AST Nodes

Figure 5.1 shows a part of the java classes that are involved with the definition of the `VariableDeclaration` class. As can be seen the AST node is an extension of the `EntityState`, this is a necessary prerequisite in order to make use of the annotations offered by the meta-model. The `VariableDeclaration` class has three fields, `ASTIdentifier`, `extraDimensions` and `initializer`.

The concrete implementation of the `VariableDeclaration` is shown in figure 5.2. As can be seen the fields of this class all carry annotations. The extra dimensions field is annotated with a simpleProperty annotation. The value of the annotation specifies of which class the object stored in the field should be an instance. In the case of the `extraDimensions` field this is the class `Integer`. Similarly, the `name` field is annotated with an `EntityProperty` annotation. This annotation also requires that the class of the field is specified as a value which must be serializable. In the case of the `name` field this is a `SimpleName` class.

5.2 Applying Changes

In the previous section, we have shown how to implement a costume `EntityState`. In this section, we show how the programmer benefits from using these annotations when applying changes in the model.

The CHA-Q meta-model defines a number of ways in which entity states can be changed. The implementation of these changes is completely defined in terms of entity states and

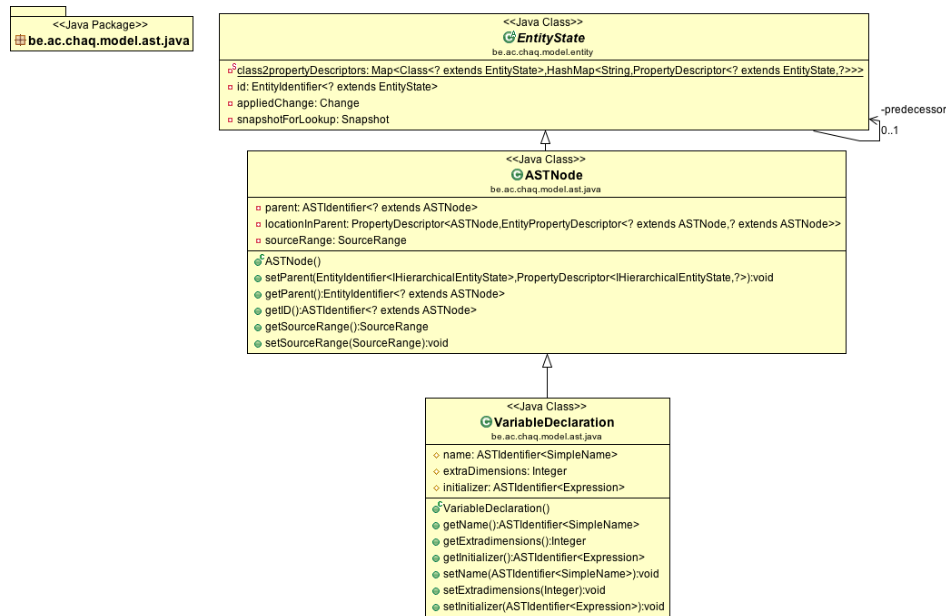

Figure 5.1: Variable declaration ASTNode

```java
public class VariableDeclaration extends ASTNode {

        @EntityProperty(value = SimpleName.class)
        protected ASTIdentifier<SimpleName> name;
        @SimpleProperty(value = Integer.class)
        protected Integer extraDimensions;
        @EntityProperty(value = Expression.class)
        protected ASTIdentifier<Expression> initializer;

        //...
}
```

Figure 5.2: Annotations in the VariableDeclaration class.

property descriptors. Because changes are defined as a high-level abstraction all the predefined and future change classes can be applied over the custom made entity states as well. In this section we show how the `VariableDeclaration` defined in the previous section can be created with a `CreationChange` and then modified with `SimplePropertyChange`. Note that these changes are predefined and offered to the programmer as part of the metamodel.

Figure 5.3 shows the creation of the variable declaration. Note that the creation change `c` is instantiated at first but the variable declaration is only added to the snapshot when the method perform is executed on the change.

Figure 5.4 shows how the `extraDimensions` of the method declaration can be changed to the value `12`. Again the change is only executed after the perform method of the change has been invoked.

```
1 TestEntityState es = new TestEntityState();
2 CreationChange c = new VariableDeclaration(rootSnapshot, es);
3 c.perform();
```

Figure 5.3: Creating and adding entity states to a snapshot.

```
1 PropertyDescriptor<TestEntityState,Integer> pd =
2 es.getPropertyDescriptorNamed("extraDimensions");
3 SimplePropertyChange spc =
4 new SimplePropertyChange(rootSnapshot, es, pd, 12);
5 EntityState newes = spc.perform();
```

Figure 5.4: Applying changes over the entity states.

As already discussed before the programmer does not need to be concerned with persistence while modelling custom entity states.

6 Evaluation

In this section, we report on an early evaluation of the CHA-Q meta-model. While the meta-model is still in development we can already successfully create a model for Java projects and deduce coarse grained changes. These coarse grained changes are currently limited to the file level. Whenever a change to a file is detected between two successive revisions of the software artefact a modified change is deduced between the two compilation units. As these changes are very coarse grained the potential reuse is much higher. When the CHA-Q meta-model is instrumented with a good change distiller it will become possible to record changes at the level of individual statements instead of on the file level. Therefore, the potential reuse will become much higher and consequently the memory and storage consumption will be reduced even farther. From experiments with such coarse changes it already became clear that the CHA-Q meta-model memory significantly reduces the memory footprint needed for storing the meta-model.

6.1 Test Data

We have evaluated the performance of the CHA-Q meta-model by applying our meta-model over a small web-application called Exapus [1]. Exapus is a web application for exploring the usage of APIs within a single project (i.e., project-centric exploration) and across a corpus of projects (i.e., api-centric exploration) along the dimensions of where, how much and in what manner [17].

The project consists of 127 compilation units, 132 class declarations and 1173 method declarations. A quick measurement showed that the memory consumption a single revision with Java JDT nodes is about 122MB. The changes in this project are very coarse grained and on average 22,5 files are modified which constitutes about one sixth of the total files in each revision. This means that the potential reuse between every iteration consists of 5/6 of the total number of files.

[1] Available online: https://github.com/cderoove/exapus

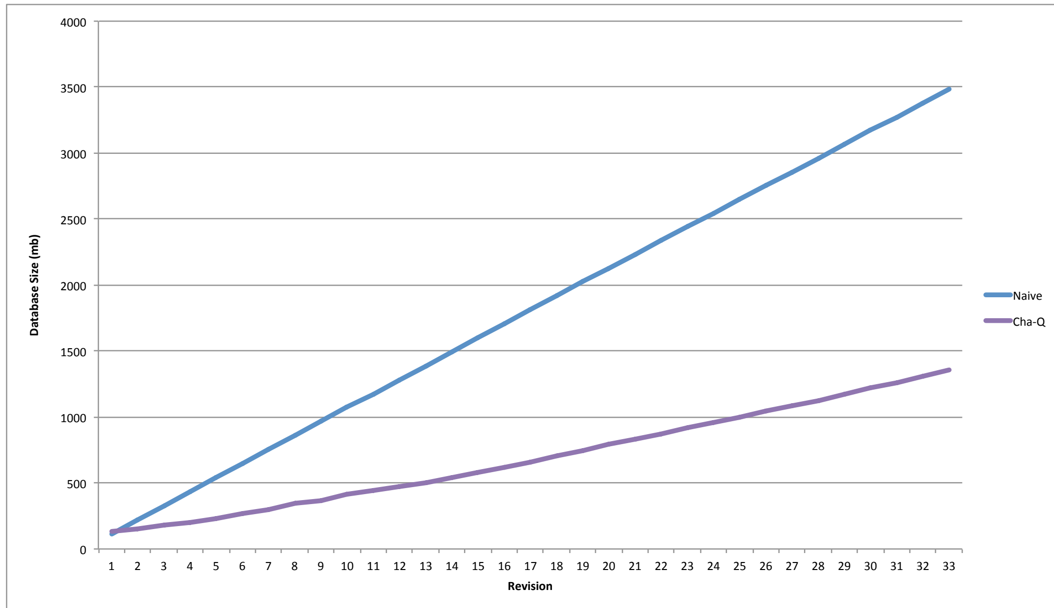

Figure 6.1: Storage overhead of the CHA-Q meta-model versus a naive implementation.

6.2 Storage and Memory Performance

In order to prove our claims that the CHA-Q meta-model can drastically reduce the memory and storage requirements we have done a number of micro benchmarks. The serialisation of a single revision consists of about 194149 nodes, 223979 properties, and 194147 relationships divided over 32 distinct relationship types. By only storing the changes between each revision the storage requirements of the CHA-Qmeta-model can be significantly reduced. We have implemented the importation and serialisation of the meta-model in two different ways. The *naive* implementation creates the meta-model as described but instead of reusing the different versions of a snapshot it creates a completely new snapshot for each revision. This implementation represent current practices which are readily available but do not optimise for storage and memory.

Because of the one-to-one mapping of the object-oriented representation to the graph database the figures for the memory requirements are very similar. The graph representation requires almost exactly the same amount of space when serialised as it requires memory in the Java VM. The performance evaluation with respect to the database storage overhead can be seen in Figure 6.1. A first observation is that our measurements are confirming the observations of prior work [9]. The naive implementation requires already over 3.4 Gb of storage space. In contrast after revision 33 the CHA-Q meta-model requires 2.5 times less storage space. While this seems far away from the 5/6 potential reuse we found that the larger files in the system were adjust more than the smaller files therefore the measurement based on file number do not give a complete image of the potential reuse. While 2.5 times less space is already a good number we are working to also feed the model with fine grained changes which should drastically increase the potential reuse and consequently decrease the storage footprint.

Conclusion

We presented the CHA-Q meta-model, a novel meta-model that provides a detailed representation of the artefacts that comprise a software system, as well as the complete history of all individual changes to these artefacts. These changes are modeled as first-class objects that can be analyzed, repeated and reverted. The Cha-Q meta-model is the first to do so for changes to artefacts other than a system's source code (e.g., bugs, bug comments, project e-mails, ...).

The meta-model supports tracking the evolution of a single entity from its creation onwards, such as the traceability link between a test case and its corresponding requirement. As such an entity is subject to changes, a means is required to uniquely identify each entity. To this end, the meta-model provides a hierarchy of identifier classes.

Applying a change results in a new state for its subject. For efficiency reasons, successive states share the values of properties that do not change. The suggested implementation strategy addresses the problem of maintaining consistency in meta-models of which all entities are interconnected transitively.

Snapshots are our meta-model's means to represent the collective state of all of a system's artefacts as seen by a particular developer at a particular point in time. Snapshots can accumulate the effect of a single or of several changes as desired —thus providing a configurable compromise between the extremes of completely version-based and completely change-based representations of a system's evolution. Revisions are modeled as snapshots that are placed under control of a version control system. The entities they are related to represent versioning information.

The meta-model currently defines classes for representing the state, evolution and changes to versioning information, bugs and source code. For the latter, the meta-model defines classes representing abstract syntax trees (modeled after the abstract grammar of the Eclipse JDT DOM for Java and the VisualWorks refactoring browser for Smalltalk) as well as classes representing object-oriented entities and their relations (modeled after the generic, language-independent FAMIX3 meta-model). Classes representing information about a system's requirements, test cases and traceability links will be defined and added to the model in the near future.

8 References

[1] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE00)*, 2000. [7, 9]

[2] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. Mse and famix 3.0: an interexchange format and source code model family. Technical report, INRIA LNE-LIRMM, 2011. [7]

[3] M. D'Ambros, H. Gall, M. Lanza, and M. Pinzger. Analysing software repositories to understand software evolution. In Tom Mens and Serge Demeyer, editors, *Software Evolution*. Springer-Verlag, 2008. [7]

[4] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution. *Journal of Software Maintenance: Research and Practice (JSME)*, 18:207–236, 2006. [7]

[5] Jannik Laval, Simon Denier, Stéphane Ducasse, and Jean-Rémy Falleri. Supporting simultaneous versions for software evolution assessment. *Science of Computer Programming*, 76(12):1177–1193, December 2011. [7, 11, 12, 13]

[6] R. Robbes and M. Lanza. Change-based software evolution. In *EVOL '06: Proceedings of the 1st International ERCIM Workshop on Challenges in Software Evolution*, pages 159–164, 2006. [7]

[7] R. Robbes and M. Lanza. Spyware: a change-aware development toolset. In *Proceedings of the 30th international conference on Software engineering (ICSE '08)*, pages 847–850. ACM, 2008. [7, 9]

[8] Peter Ebraert. *A bottom-up approach to program variation*. PhD thesis, Vrije Universiteit Brussel, March 2009. [7, 9]

[9] Lile Palma Hattori. *Change-centric Improvement of Team Collaboration*. PhD thesis, Università della Svizzera Italiana, 2012. [7, 9, 11, 12, 24]

[10] Maximilian Koegel, Jonas Helming, and Stephan Seyboth. Operation-based conflict detection and resolution. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models (CVSM09)*, 2009. [7, 9]

[11] Markus Herrmannsdoerfer and Maximilian Koegel. Towards a generic operation recorder for model evolution. In *Proceedings of the 1st International Workshop on Model Comparison in Practice (IWMCP10)*, pages 76–81, 2010. [7]

[12] Katsuhisa Maruyama, Eijiro Kitsu, Takayuki Omori, and Shinpei Hayashi. Slicing and replaying code change history. In *Proceedings of the 27th nternational Conference on Auto-*

mated Software Engineering (ASE12), pages 246–249, 2012. [8]

[13] H. Gall, B. Fluri, and M. Pinzger. Change analysis with evolizer and changedistiller. *IEEE Softw.*, 26(1):26–33, 2009. [8, 9]

[14] Dane Marjanovic. Developing a meta model for release history systems. Master's thesis, University of Zurich, 2006. [8]

[15] Verónica Uquillas Gómez. *Supporting Integration Activities in Object-Oriented Applications*. PhD thesis, Vrije Universiteit Brussel - Université des Sciences et Technologies de Lille, October 2012. [9, 11, 12, 13]

[16] Martin Pinzger and Harald C. Gall. Dynamic analysis of communication and collaboration in oss projects. In *Collaborative Software Engineering*, pages 265–284. Springer Berlin Heidelberg, 2010. [9]

[17] C. De Roover, R. Lammel, and E. Pek. Multi-dimensional exploration of api usage. In *21st International Conference on Program Comprehension (ICPC)*, pages 152–161, 2013. [23]