



Commutative Event Sourcing vs. Triple Graph Grammars

Sebastian Copei and Albert Zündorf

Kassel University, Germany

Abstract. This paper proposes Commutative Event Sourcing as a simple and reliable mechanism for model synchronisation, bidirectional model to model transformations, incremental updates, and collaborative editing. Commutative Event Sourcing is a restricted form of a Triple Graph Grammar where the **rules or editing commands** are either overwriting or commutative. This restriction gets rid of a lot of Triple Graph Grammar complexity and it becomes possible to implement model synchronisation directly. You are not restricted to Java as your programming language and you do not need to use a proprietary library, framework, or tool.

Keywords: Event Sourcing · Triple Graph Grammars · Bidirectional Model Transformations.

1 Introduction

Whenever you have two tools that each deploy their own meta model but interchange related or overlapping data you face the problem of model synchronisation: whenever some common data is modified in one tool, you want to update the corresponding data in the other tool. Note, when both models use the same meta model, the problem of model synchronisation becomes closely related to model versioning and to the merging of concurrent model changes.

In the area of bidirectional (BX) transformations there are various approaches attacking the problem of model synchronisation cf. [6,3]. Among the various approaches, we consider Triple Graph Grammars (TGGs) [14] to be the most mature and most practical solution with a lot of tool support [10]. Recent development in TGG tools provide support for incremental model synchronisation [11], i.e. the effort for model synchronisation is proportional to the model change performed. In [9] Fritsche et al present recent advances in achieving incremental model synchronisation.

While TGGs have a lot of mature tool support and a very sound theory, it is quite complex to implement a TGG tool and to apply a TGG approach within your own application. You will probably fail to implement your own approach and you will need to use some existing tool that requires you to adopt a lot of tool specific prerequisites (e.g. the Eclipse Modeling Framework [15]) and to learn a lot about (triple) graph grammar theory in order to write down appropriate TGG rules.

This paper proposes Commutative Event Sourcing (CES) as an alternative approach to model synchronisation. As we will show, CES is equivalent to a restricted form of TGGs where the order of rule application is commutative. This facilitates the implementation of CES tremendously, such that you may adopt our approach without the need of using a proprietary tool and without graph grammar theory. You just follow some design patterns that we propose and implement your own incremental model synchronisation manually.

2 Triple Graph Grammars (TGGs)

This section revisits the work of Fritsche et al [9]. The running example of [9] and of this paper is the synchronisation of Java package and Java class models with JavaDoc folders and files. Figure 1 shows the class diagrams for the two models as used in our implementation of this example.

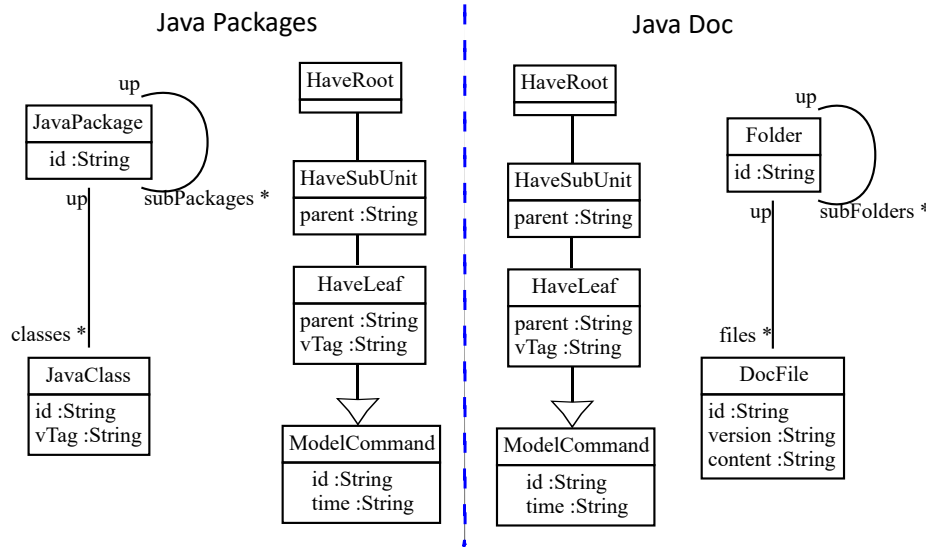


Fig. 1. Classes for Java and JavaDoc structures plus edit commands

The left of Figure 1 shows the class model for the Java packages tool. Basically there is the class **JavaPackage** that may have multiple **subPackages**. In addition, a **JavaPackage** may have many classes of type **JavaClass**. Our class model extends the example from [9] with **id** attributes and with a **vTag** attribute. The latter will be used to discuss some editing or merge conflicts that are not handled by [9]. The **id** attributes are used for referencing across tools. The **id** attributes are the first means that we introduce in order to facilitate the model synchronisation task.

The left of Figure 1 also shows the `ModelCommand` classes `HaveRoot`, `HaveSubUnit`, and `HaveLeaf`. These classes are part of our CES approach and will be discussed in Section 4. The right of Figure 1 shows the classes `Folder` and `DocFile` with the associations `subFolders` and `files` that model the JavaDoc structures. Again we have the same `ModelCommand` classes as for Java packages.

Figure 2 shows a slight modification of the Triple Graph Grammar rules used in [9] to solve the model synchronisation problem for our Java to JavaDoc example. There are a `HaveRoot`, a `HaveSubUnit`, and a `HaveLeaf` rule. Each rule shows its name and its parameters in a hexagon in the middle of the rule. Each rule has a left and a right subrule. Each subrule specifies three possible operations: `run`, `undo`, and `parse`. We will walk through these operation types one by one.

The `run` operation for a subrule tries to create the specified situation. Thus, all green parts of the subrule are going to be created, all black parts are required to already exist, and all blue parts must not exist (and may need to be removed by the `run` operation). Thus, the left subrule of the `HaveRoot` rule describes that on execution a `JavaPackage` object `p` shall be created in the Java model. The `id` attribute of `p` is copied from the `id` parameter of the rule. The blue parts of the `HaveRoot` rule require that there must be no `JavaPackage` `sp` attached to `p` via an `up` link. Thus we shall reset this link. Our manual implementation of this subrule is shown in Listing 1.1 See [1] for a complete reference.

```

1 package JavaPackages;
2 public class HaveRoot extends ModelCommand {
3     @Override
4     public Object run(JavaPackagesEditor editor) {
5         JavaPackage p = (JavaPackage) editor
6             .getOrCreate(JavaPackage.class, getId());
7         p.setUp(null);
8         return p;
9     }
10    ...

```

Listing 1.1. Manual implementation of `HaveRoot` rule for Java packages

The `run` method of our `HaveRoot` command uses an editor to `getOrCreate` the desired `JavaPackage` object. Our editor maintains a hash table storing `id` object pairs, cf. Section 5. This hash table is e.g. used in the `HaveSubUnit` command to look up the required `sp JavaPackage`, cf. method `getObjectFrame` in Line 8 of Listing 1.2 and Section 5. Note, in the left subrule of the `HaveSubUnit` rule of Figure 2 the upper `JavaPackage` `sp` is shown in black color. This means the execution of the subrule requires that `sp` exists as context for the creation of the sub package `p`. The green `up` link requires that `p` needs to be connected to `sp` via an `up` link (which simultaneously creates the `subPackages` link in the reverse direction).

```

1 package JavaPackages;
2 public class HaveSubUnit extends ModelCommand {

```

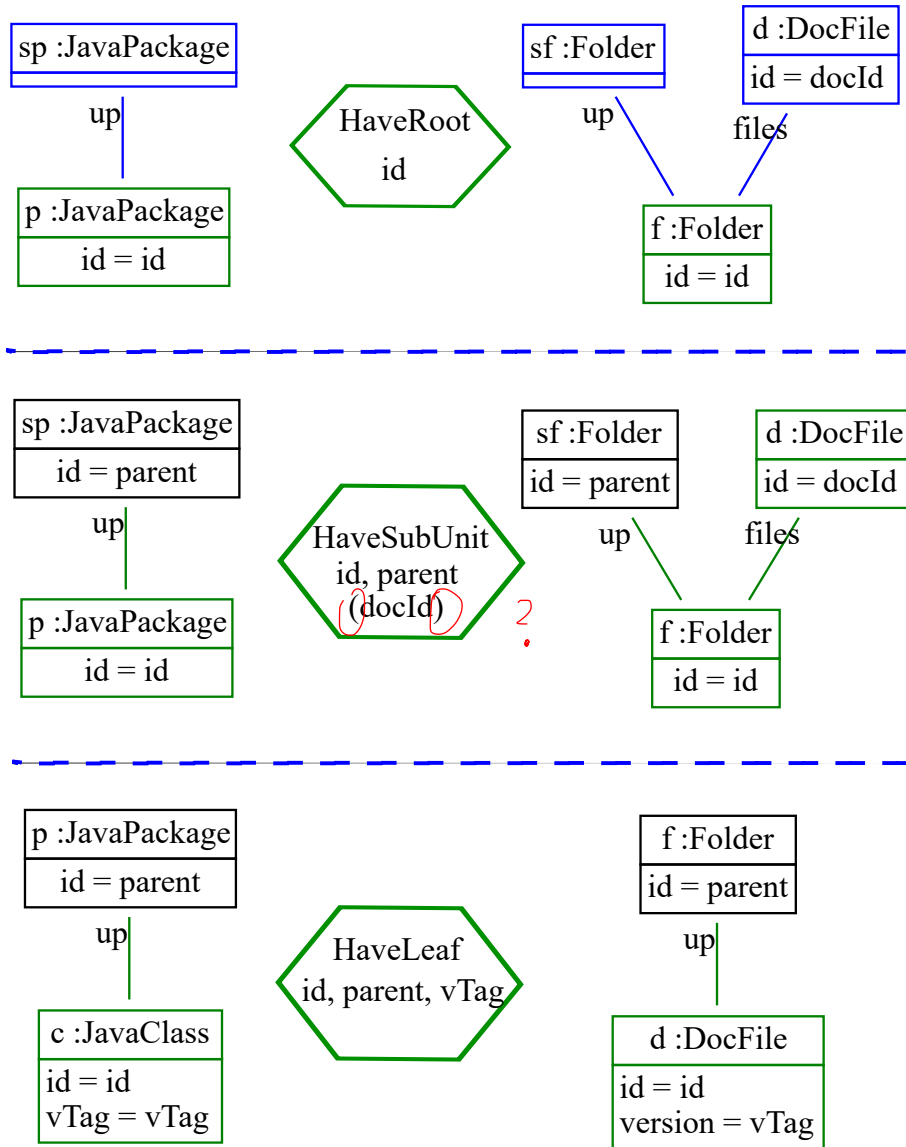




Fig. 2. Triple Graph Grammar (like) rules for Java and JavaDoc structures



```

3      @Override
4      public Object run(JavaPackagesEditor editor) {
5          JavaPackage p = (JavaPackage) editor
6              .getOrCreate(JavaPackage.class, getId());
7          JavaPackage sp = (JavaPackage) editor
8              .getObjectFrame(JavaPackage.class, this.parent);
9          p.setUp(sp);
10         return p;
11     }
12     ...

```



Listing 1.2. Manual implementation of HaveSubUnit rule for Java packages

For completeness, Listing 1.3 shows the HaveLeaf command.

```

1  package JavaPackages;
2  public class HaveLeaf extends ModelCommand {
3      @Override
4      public Object run(JavaPackagesEditor editor)
5      {
6          JavaClass c = (JavaClass) editor
7              .getOrCreate(JavaClass.class, getId());
8          JavaPackage p = (JavaPackage) editor
9              .getObjectFrame(JavaPackage.class, this.parent);
10         c.setUp(p);
11         c.setVTag(this.vTag);
12         return c;
13     }
14     ...

```

Listing 1.3. Manual implementation of HaveLeaf rule for Java packages

In Listing 1.4 we create a number of command objects and initialize their parameters, appropriately. Then we ask an appropriate editor to execute the commands. The editor adds the commands to its command history and then calls their run method, cf. Section 4. This results in the object structure shown on the left of Figure 3.

```

1  package javaPackagesToJavaDoc;
2  ...
3  public class TestPackageToDoc extends ModelCommand {
4      private void startSituation(JavaPackagesEditor editor) {
5          ModelCommand cmd = new HaveRoot().setId("root");
6          editor.execute(cmd);
7          cmd = new HaveSubUnit().setParent("root").setId("sub");
8          editor.execute(cmd);
9          cmd = new HaveSubUnit().setParent("sub").setId("leaf");
10         editor.execute(cmd);
11         cmd = new HaveLeaf().setParent("leaf")

```

```

12         .setVTag("1.0").setId("c");
13     editor.execute(cmd);
14 }
15 ...

```

Listing 1.4. Invoking triple rules or commands

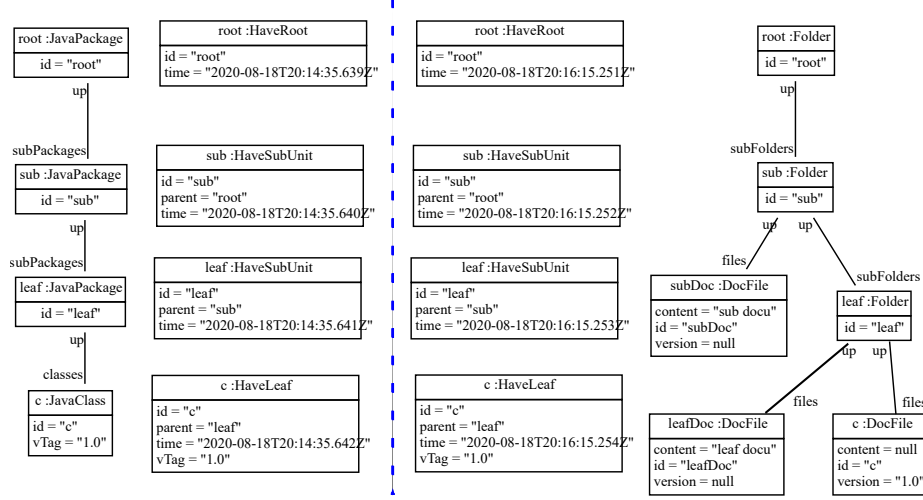


Fig. 3. Objects for Java and JavaDoc structures plus edit commands

Listing 1.5 shows our manual implementation of the **HaveRoot** command for JavaDoc structures. This implements the right sub rule of the **HaveRoot** triple rule of Figure 2. Lines 5 to 7 of Listing 1.5 are quite similar to the corresponding **JavaPackages** command. They just create a **Folder** instead of a **JavaPackage**. Lines 8 to 13 of Listing 1.5 remove a potentially existing **DocFile** *d*. Such an object *d* might have been created by previous command executions. The blue parts of the right subrule of our **HaveRoot** triple rule require that such a **DocFile** must not exist. Listing 1.6 and Listing 1.7 show the manual implementation of the other two **JavaDoc** subrules.

```

1 package JavaDoc;
2 public class HaveRoot extends ModelCommand {
3     @Override
4     public Object run(JavaDocEditor editor) {
5         Folder f = (Folder) editor
6             .getOrCreate(Folder.class, getId());
7         f.setUp(null);
8         String docId = getId() + "Doc";
9         DocFile d = obj.getFromFiles(docId);

```

```

10      if (d != null) {
11          editor.removeModelObject(d.getId());
12          f.withoutFiles(d);
13      }
14      return f;
15  }
16  ...

```

Listing 1.5. Manual implementation of HaveRoot rule for JavaDoc

```

1  package JavaDoc;
2  public class HaveSubUnit extends ModelCommand {
3      @Override
4      public Object run(JavaDocEditor editor) {
5          Folder f = (Folder) editor
6              .getOrCreate(Folder.class, getId());
7          Folder sf = (Folder) editor
8              .getObjectFrame(Folder.class, parent);
9          f.setUp(sf);
10         String docId = getId() + "Doc";
11         DocFile d = (DocFile) editor
12             .getOrCreate(DocFile.class, docId);
13         d.setContent(getId() + "_docu");
14         f.withFiles(d);
15         return f;
16     }
17     ...

```

Listing 1.6. Manual implementation of HaveSubUnit rule for JavaDoc

```

1  package JavaDoc;
2  public class HaveLeaf extends ModelCommand {
3      @Override
4      public Object run(JavaDocEditor editor) {
5          DocFile d = (DocFile) editor
6              .getOrCreate(DocFile.class, getId());
7          Folder f = (Folder) editor
8              .getObjectFrame(Folder.class, parent);
9          d.setUp(f);
10         d.setVersion(vTag);
11         return d;
12     }
13     ...

```

Listing 1.7. Manual implementation of HaveLeaf rule for JavaDoc

We might invoke the **JavaDoc** commands as we have done this for the **JavaPackages** in Listing 1.4. Alternatively, in Line 12 of Listing 1.8 we lookup the list if com-

mands that our `javaPackagesEditor` has collected while building the start situation. Then Line 13 uses a simple `Yaml` encoder to serialize these commands into a string in `Yaml` format. You may use any `JSON` based serialization, either. The serialization task is pretty simple, as our commands use string based parameters only and have no references to other objects. Finally, Line 14 calls method `loadYaml` on the `javaDocEditor`. Method `loadYaml` turns the passed string into `JavaDoc` command objects and executes these. The result is shown on the right of Figure 3.

```

1 package javaPackagesToJavaDoc;
2 ...
3 public class TestPackageToDoc extends ModelCommand {
4     @Test
5     public void testFirstForwardExample()
6     {
7         JavaPackagesEditor javaPackagesEditor =
8             new JavaPackagesEditor();
9         startSituation(javaPackagesEditor);
10        JavaDocEditor javaDocEditor = new JavaDocEditor();
11        Collection commands = javaPackagesEditor
12            .getActiveCommands().values();
13        String yaml = Yaml.encode(commands);
14        javaDocEditor.loadYaml(yaml);
15        ...
16    }
17    ...

```

Listing 1.8. Invoking triple rules or commands

In a Triple Graph Grammar (TGG) tool based approach, you would just provide the triple rules shown in Figure 2. Then the corresponding commands would be derived using either a code generator or a TGG interpreter. See [1] for an example of such an interpreter. Using a TGG tool, you get not only the forward execution of rules but also **undo** and **parse** functionality. Undoing a TGG subrule basically requires to remove all model parts that have been created for green rule elements on the forward execution. In a manual implementation, you have to implement the **undo** step yourself and it must be consistent to the **run** operation. Listings 1.9, 1.10, and 1.11 show our manual implementation of the **undo** operations for the `JavaPackages` commands. For the `JavaDoc` commands see [1].

```

1 package JavaPackages;
2 public class HaveRoot extends ModelCommand {
3     ...
4     @Override
5     public void undo(JavaPackagesEditor editor)
6     {
7         editor.removeModelObject(getId());

```




```

8     }
9     ...

```

Listing 1.9. Manual implementation of HaveRoot.undo() for Java packages

```

1 package JavaPackages;
2 public class HaveSubUnit extends ModelCommand {
3     ...
4     @Override
5     public void undo(JavaPackagesEditor editor)
6     {
7         JavaPackage obj = (JavaPackage) editor
8             .removeModelObject(getId());
9         obj.setUp(null);
10    }
11    ...

```

Listing 1.10. Manual implementation of HaveSubUnit.undo() for Java packages

```

1 package JavaPackages;
2 public class HaveLeaf extends ModelCommand {
3     ...
4     @Override
5     public void undo(JavaPackagesEditor editor)
6     {
7         JavaClass obj = (JavaClass) editor
8             .removeModelObject(getId());
9         obj.setUp(null);
10    }
11    ...

```

Listing 1.11. Manual implementation of HaveLeaf.undo() for Java packages

Usually, TGG rule applications depend on each other. For example if we call `undo` on the `JavaPackages HaveRoot` command of our example, this would remove the `root JavaPackage` from our model, cf. Figure 3. This would leave the `sub JavaPackage` without a parent. Thus the `HaveSubUnit` TGG rule does no longer match for the `sub` object. This is important as on model synchronisation the right subrule of `HaveSubUnit` creates the `subDoc DocFile` which is no longer valid. To repair this, a standard TGG approach has to `undo` dependant rule applications whenever their context becomes invalid. Thus on `undo` of the `HaveRoot` command, a standard repair step would also `undo` the `HaveSubUnit` command for the `sub JavaPackage` and in turn `undo` the commands for the `leaf` and for the `c` objects. Via model synchronisation all `JavaDoc` objects will be removed, either. To keep the lower model parts, one would apply a `HaveRoot` command on `sub` and rerun the `HaveSubUnit` and `HaveLeaf` commands on `leaf` and `c`. Formally, the whole model would be deleted and reconstructed. In [9] this is called a cascading delete and [9] proposes sophisticated theory and means to

avoid this cascading delete via so called short-cut-repair rules. In our manual implementation it would suffice to **run** a **HaveRoot** command on **sub** in order to repair the situation. This is achieved as Line 7 of Listing 1.1 and Lines 7 to 13 of Listing 1.5 carefully remove all model parts that correspond to the blue parts of the **HaveRoot** TGG rule, cf. Figure 2, i.e. all model parts that might stem from a previous application of a **HaveSubUnit**. In a manual implementation you have to spot the overlap of the **HaveRoot** and the **HaveSubUnit** rules yourself and you have to design these rules and their manual implementation very carefully in order to circumvent cascading deletes. Commutative Event Sourcing will help you to achieve this as discussed in Sections 3 and 4. [9] does this for you automatically which is a really great job.


Whenever you edit a model directly without using the TGG rules or the corresponding commands and you want to do a new model synchronisation, you need to parse the changed model in order to identify which TGG rule applications are now valid. Again TGG tools do this parsing for you. Basically all green and black parts of a TGG subrule must be matched and the blue parts must not be there. In general TGG parsing has to deal with rule dependencies, too. Usually, TGG parsing needs to find all applications of so-called "root" rules (that do not depend on other rules), first. Then TGG parsing, inspects the surroundings of "root" rule applications and try to find applications of rules that use the "root" rules in their context. In turn, you apply rules which context has then become available. In our example this results in some kind of top-down parsing starting with the **root** **JavaPackage** and descending to **subPackages** and **classes**, recursively. In general, TGG parsing may be even more complicated, cf. [14]. Again Commutative Event Sourcing allows us to facilitate parsing considerably as we will discuss in Section 6.

Thus in our manual implementation we ignore the order of rule applications for now. For a single rule, parsing is relatively simple. Listing 1.12 shows the manual implementation of the **parse** method for our **HaveRoot** command for **JavaPackages**. Our editor calls the **parse** methods of our commands when appropriate, cf. Section 6. On such a call, the editor passes an object that may have been modified and needs parsing as parameter. Thus Line 6 of the **parse** method of Listing 1.12 first ensures that the current object is a **JavaPackage**. Similarly, Line 10 ensures that there is no upper **JavaPackage**. If the current package has no sub packages and contains no **JavaClasses**, we consider it garbage. Therefore, Line 15 to 17 return a **RemoveCommand** with the corresponding object **id**. Without such a garbage collection mechanism, the users would have to invoke **RemoveCommands** manually in order to get rid of model objects. Finally Line 20 to 22 create a **HaveRoot** command and retrieve its **id** parameter from the model and return it.

```

1 package JavaPackages;
2 public class HaveRoot extends ModelCommand {
3     ...
4     @Override
5     public ModelCommand parse(Object currentObject) {

```



```

6      if (! (currentObject instanceof JavaPackage)) {
7          return null;
8      }
9      JavaPackage currentPackage = (JavaPackage) currentObject;
10     if (currentPackage.getUp() != null) {
11         return null;
12     }
13     if (currentPackage.getClasses().isEmpty()
14         && currentPackage.getSubPackages().isEmpty()) {
15         ModelCommand modelCommand = new RemoveCommand()
16             .setId(currentPackage.getId());
17         return modelCommand;
18     }
19     // yes its me
20     ModelCommand modelCommand = new HaveRoot()
21         .setId(currentPackage.getId());
22     return modelCommand;
23 }
24 ...

```

Listing 1.12. Manual implementation of the parse step for HaveRoot for Java packages

Listing 1.13 shows the `parse` method of the `HaveSubUnit` command for `JavaPackages`. Note, that this method is slightly simpler as the garbage collection is done by the `HaveRoot` command. You find the `parse` method of the `HaveLeaf` command for `JavaPackages` in [1]. We leave the implementation of the parse methods for the `JavaDoc` commands as an exercise for the interested reader.

```

1  package JavaPackages;
2  public class HaveSubUnit extends ModelCommand {
3      ...
4      @Override
5      public ModelCommand parse(Object currentObject) {
6          if (! (currentObject instanceof JavaPackage)) {
7              return null;
8          }
9          JavaPackage currentPackage = (JavaPackage) currentObject;
10         if (currentPackage.getUp() == null) {
11             return null;
12         }
13         ModelCommand modelCommand = new HaveSubUnit()
14             .setParent(currentPackage.getUp().getId())
15             .setId(currentPackage.getId());
16         return modelCommand;
17     }

```

18 . . .


Listing 1.13. Manual implementation of the parse step for `HaveSubUnit` for Java packages

3 Commutative Event Sourcing (CES) Theory

Event Sourcing has been proposed by [8] and [17] as a means for communication between multiple domains or (micro) services. Basically, a program or service logs relevant operations as events and these events are then transferred to other programs or services that react with appropriate operations on their site. In order to use this idea for model synchronisation, we just log all editing operations / commands on one model and then send these editing events to some other model and perform similar changes there. This is clearly related to the Triple Graph Grammar approach discussed in Section 2. However, Event Sourcing has some additional requirements that ultimately lead us to Commutative Event Sourcing.

To connect multiple applications, Event Sourcing is frequently based on some message broker mechanism. Depending on the quality of service that your message broker provides, messages may be lost or received in wrong order and messages may be received multiple times. For example, there may be a (`HaveRoot root`) event and a (`HaveSubUnit sub root`) event and for some reasons you may receive only the latter. In [13] we deal with this problem by adding `ids` to all events / commands and by extending each event with the `id` of its predecessor. Thus if you receive event (`#2 HaveSubUnit sub root #1`) and you did not yet receive event `#1` you postpone the execution of the `#2` event and ask for re-submission of event `#1`. Event `ids` also allow to detect duplicated receipt of the same event. However if there are two independent tools that raise e.g. event (`#3 HaveLeaf c sub 1.0 #2`) and (`#3 HaveLeaf c sub 1.1 #2`) you have a merge conflict and if you can not rely on the order of arrival, there is no reliable way to resolve this merge conflict. In general, it is tricky to get a linear event numbering if there are concurrent edits in multiple tools. Therefore, [13] uses timestamps instead of running numbers as event identifiers. This allows to resolve merge conflicts deterministically but requires a reasonable clock synchronisation between the tools.

In our approach, it is possible to execute a `HaveSubUnit sub root` command before the required `root` is created. We achieve this by using `ids` for model objects and by having a hash table of all model objects and by using `getOrCreate` and `getObjectFrame` operations that do a hash table lookup for the required object (`id`) and create the object if it is missing, cf. Line 8 of Listing 1.2 and Section 6. Actually, you can execute any command or TGG subrule as soon as you have enough information to establish the required context. For our `HaveSubUnit` commands this means, you need to know the `id` and the class of the required upper `JavaPackage`. However, a later execution of the (`HaveRoot root`) command must use the same `getOrCreate` mechanism in order to glue the results of these two edits together. If all your commands have sufficient parameters and information to establish their context themselves and to glue their own new objects

to the context of other commands, commands may be executed in any order, i.e. commands become commutative. Having commutative commands gets rid of the effort for command ordering. And it facilitates parsing, cf. Section 6. This lead to our idea of Commutative Event Sourcing. 

Next, we provide some theory that specifies the requirements for Commutative Event Sourcing. Section 4 then shows simple patterns that achieve a sound implementation of Commutative Event Sourcing. A previous version of our theory has been published in [4]. However, this paper restructures our theory in large parts and it even enriches our formal requirements in order to further facilitate the implementation of a Commutative Event Sourcing application.

First let us set up some basic notations: like [16] we use capital letters such as M , N for metamodels i.e. for sets of models (that adhere to a common class diagram). \emptyset denotes an empty model. We denote events with e and the set of all possible events with E . An event $e = (t, i, x_1, \dots, x_n)$ has an event type $t \in T$, an event identifier $i \in CHAR^*$ (i.e. some string), and a number of parameters $x_i \in R \cup CHAR^*$, i.e. parameters are arbitrary numbers or strings. We will also use series of events $\bar{e} = (e_1, \dots, e_n)$ and denote by \bar{E} the set of all possible event series with events from E . Similarly we use sets of events $\tilde{e} = \{e_1, \dots, e_n\} \in \mathcal{P}(E)$.

Events may be applied to (or synchronized with) models via function $apply(e, m)$, which generates a possibly new model m' . Furthermore, we define the application of an event series $\bar{e} = (e_1, \dots, e_n)$ to a model m as $apply(\bar{e}, m) = apply(e_n, \dots, apply(e_2, apply(e_1, m)) \dots)$.


Now, we want a simple mechanism that allows us to restrict the size of our command histories. Therefore, Definition 2 requires that all identifiers in a series of effective events must be unique. This allows us to maintain our commands within a hash table and if we run a new command with an already used id, Definition 2 allows us to overwrite the old hash table entry in our command history with the new command.


Definition 1. (*effective events*) An event e_p is called *ineffective* within an event series \bar{e} at position p iff $apply(\bar{e} \setminus e_p, \emptyset) = apply(\bar{e}, \emptyset)$.

We call e_p *effective*, otherwise.

We call an event series \bar{e} *effective* iff it contains only effective events.

Definition 2. (*overwriting*) We call **events** *overwriting*, if for all pairs of events e_1, e_2 with identifiers i_1 and i_2 contained in **an** effective event series \bar{e} holds $i_1 \neq i_2$

Next we want to get rid of command dependencies, i.e. we want to be able to execute a command as soon as we receive it without waiting for other commands to establish a required context. Similarly, we would like to store our command hash table persistently and to reload and rerun all commands on tool start up without bothering with command order. 

Definition 3. (*commutativity*) We call events *commutative*, if for all models $m \in M$ and all events $e_1, e_2 \in E$ holds that $apply(e_2, apply(e_1, m)) = apply(e_1, apply(e_2, m))$ 

Definition 3 just states that command execution must be commutative. Section 4 will show how to achieve this in your implementation (even for overwriting events, spoiler: e.g. via time stamps). Here we use Definition 2 and Definition 3 to define a minimal set of events \tilde{e} that is derived from an event series \bar{e} by removing all events e that are ineffective and then collecting the remaining events within a set.

We are now ready to define M_{CES} the set of all models that may be created via Commutative Event Sourcing:

Definition 4. (M_{CES}) *A model m is supporting Commutative Event Sourcing if and only if there exists a **minimal** event set \tilde{e} with $\text{apply}(\tilde{e}, \emptyset) = m$ and a function $\text{parse} : M \rightarrow \mathcal{P}(E)$ such that $\text{parse}(m) = \tilde{e}$. We say $m \in M_{CES}$.*

This means, there must be a bidirectional mapping between the command history and its model and it must be possible to reconstruct the command history from the model through parsing. Now we are able to define whether two models are "equivalent" or synchronized. Two models are synchronized if their minimal event sets are equal. We may also opt for partial synchronisation e.g. we may restrict ourselves to certain event types. Then two models are partially synchronized if their minimal event sets restricted to the desired event types are equal. This allows e.g. to have additional **HaveContent** commands in our **JavaDoc** tool that add the actual content to our **JavaDoc Files**. This additional information is not synchronized with our **JavaPackages** tool, cf. [1].

Overall, we restrict ourselves to commands that have to be implemented in a very specific way. However, this should not restrict the kind of models that may be generated, too severe. Generally, given a class diagram or metamodel, you may introduce one command for each pair of class and attribute and one command for each association. Then all you need are unique identifiers for each object and you are able to use Commutative Event Sourcing. However, if you use CES for model synchronisation between different meta models, such a simple "one command per attribute" approach may not work and you may need slightly more complex commands that create "similar" things in both models. This means, for model synchronisation you need to come up with commands that serve both models. If you feel that this couples the design of your tools too strictly, you may think about the construction of an anti corruption layer [8].

4 Achieving Commutative Event Sourcing

This section provides some simple design rules to achieve Commutative Event Sourcing. These design rules are based on the notion of an *increment*. An increment is a set of attributes and links that are edited through a single command execution. To achieve commutative commands, we just split each model into a disjoint set of increments. Thus, each attribute and link of a given model is edited by exactly one command execution. Actually, two commands may edit the same attribute, iff they assign the same value to it. We use this e.g. for **id** attributes.

In addition, object creation is done lazily, via `getOrCreate` operations as described in Section 5. Finally, each increment must contain sufficient information in order to reconstruct the command (and its parameters) that created it.

As a rule of thumbs, we design the commands for model synchronisation by going through the class diagrams of the models to be synchronised and group all attributes and associations into command groups. Then we implement the corresponding commands such that they assign defined values to all attributes and links that belong to their group. In case of the `JavaPackages` class diagram shown in Figure 1 the `id` attribute of class `JavaPackage` and the `subPackages` - `up` association form such a group for our `HaveRoot` and our `HaveSubUnit` commands. Similarly, the `id` attribute and the `vTag` attribute of class `JavaClass` and the `classes` - `up` association form such a group for our `HaveLeaf` command. Accordingly, the `run` methods of the `HaveRoot` and `HaveSubUnit` commands both assign values to the `id` attributes of their `JavaPackage` objects by calling `getOrCreate`, cf. Line 6 of Listing 1.1 and Line 6 of Listing 1.2. Similarly, both commands call `setUp`, cf. Line 7 of Listing 1.1 and Line 9 of Listing 1.2. Note, the values of the `id` attribute and the `up` attribute also suffice to reconstruct the command used on them, cf. Listing 1.12 and Listing 1.13. Similarly, the `JavaPackage` implementation of the `HaveLeaf` command takes care of all attributes and links in its increment, cf. Listing 1.3 and Listing 1.11.

Note, for the `JavaDoc` versions of these commands, the case is more complicated. Here a `HaveSubUnit` commands does not only create a `Folder` and an `up` link but also a `DocFile` with a certain `id`, cf. Listing 1.6. Similarly, the `JavaDoc HaveRoot` command takes care of such a `DocFile`, cf. Listing 1.5.

In case of a manual implementation of the commands, you may check their commutativity by applying a sufficiently long series of commands once in order and once in reverse order and then you compare the resulting models. To our experiences, this reveals violations of the commutativity rule, quite reliably. Note, due to our excessive use of `ids`, comparison of two models is quite easy in our case.

If you use TGG rules, your rules should use some `getOrCreate` mechanism to create overlapping (context) parts only once and they should have the `Local Church Rosser property`, cf. [12]. For certain graph grammars this property may be checked at compile time. Thus, for Triple Graph Grammars you may prove that your rules are commutative and do not need to rely on testing. Just note, some additional care is needed if two rules edit the same increment. Then these rules must be proven to be `overwriting`, i.e. to edit all parts of that increment.

Once you got your commands right, we still have to deal with multiple or concurrent edits of the same increment. Assume, you have already executed command (`HaveLeaf c sub 1.0`) and now you want to change the version to 1.1. Thus you run the command (`HaveLeaf c sub 1.1`). In your current tool the second command would just overwrite the first command and everything is fine. Unfortunately, model synchronisation may fail, if your message broker delivers theses two commands in the wrong order. Similarly, you have a model synchronisation problem, if the two commands are run in two different tools,

concurrently, and you try to synchronize afterwards. As the two commands are conflicting, i.e. they assign different values to the `vTag` attribute of `JavaClass c`, model synchronisation needs some mechanism that decides which command overwrites the other.

This problem is not yet addressed by the Triple Graph Grammar tools we are aware of. Actually, [9] explicitly mentions this problem and puts it in future work. This paper solves this problem with the help of additional time stamps. Listing 1.14 shows the `execute` method that we deploy in our editors.

```

1 package JavaPackages;
2 public class JavaPackagesEditor {
3     ...
4     public void execute(ModelCommand command) {
5         String id = command.getId();
6         if (id == null) {
7             id = "obj" + activeCommands.size();
8             command.setId(id);
9         }
10        String time = command.getTime();
11        if (time == null) {
12            time = getTime();
13            command.setTime(time);
14        }
15        ModelCommand oldCommand = activeCommands.get(id);
16        if (oldCommand != null) {
17            if (oldCommand.getTime().compareTo(time) > 0) {
18                return;
19            } else if (oldCommand.getTime().equals(time)) {
20                String oldYaml = Yaml.encode(oldCommand);
21                String newYaml = Yaml.encode(command);
22                if (oldYaml.compareTo(newYaml) >= 0) {
23                    return;
24                }
25            }
26        }
27        command.run(this);
28        activeCommands.put(id, command);
29    }
30    ...

```

Listing 1.14. Command execution

When you want to execute a command you actually call `editor.execute(cmd)` on the responsible editor. Line 6 of Listing 1.14 first checks if your command has an `id`. If not, we create an `id` for you. This facilitates testing and iterative development. Alternatively, you may raise an exception. Next, Line 10 reads the time stamp of the command. If there is no time stamp yet, we assign the current

time, cf. Line 13¹. If you serialize a command later on and send it e.g. to another tool, the command will already contain the original time stamp.

Now Line 15 does a lookup in our hash table for `activeCommands`. If we already have an `oldCommand` (Line 16) and the time stamp of that `oldCommand` is greater than the time stamp of the new `command` (Line 17), then we do not execute the new `command`, cf. Line 18. If the time stamps are equal (Line 19) we retrieve the yaml representation of the two commands and if the yaml representation of the `oldCommand` is greater equal the new command (Line 22), we again ignore the new command. Note this handles the case that a command is executed twice and the unlikely event that two tools create a command with the same id and the same time stamp, independently. In the latter case, we just keep the lexically later command. If it is a new `command`, the editor executes it in Line 27. Finally, the new `command` is added to the hash table of active commands.

Thus, if you run (`HaveLeaf c sub 1.0`) in some editor at e.g. 13:36 o'clock and you run (`HaveLeaf c sub 1.1`) at e.g. 13:37 o'clock on the same editor, the second command will overwrite the first. If you run the two commands on two different editors, each editor will store its own command. If you do model synchronisation some time later, on the first editor the 13:37 command will overwrite the 13:36 command while on the other editor the 13:36 command will be ignored as that editor already has the 13:37 command for the same id. Eventually, both editors have the 13:37 command active and the `vTag` of `c` will be 1.1. Note, this model synchronisation scheme works also for two editors with different meta models.



5 Removing model objects

There is yet another design issue to be discussed. Due to our `getOrCreate` mechanism, removing an object from our model is quite tricky. To safely remove an object from our model we must get rid of the command that created and initialized it directly (green parts of our TGG rules, cf. Figure 2). And we must get rid of all usages of this object as context (black rule part) in all other commands. Generally, we would require that for any object that is used as a required context (black rule part) by some command the corresponding set of `activeCommands` shall contain another command that explicitly creates and initializes that object (green rule part). Unfortunately, we deal with unreliable message brokers and some commands may just not yet have arrived. But we already want to work with our model. Thus, it would be great to be able to distinguish between fully initialized *model objects* and so-called *object frames* that so far have been used as context objects, only. To achieve this, our editors deploy one `mapOfModelObjects` for explicit model objects and one `mapOfFrames` for context objects, cf. Listing 1.15 and [1]. (The `mapOfParsedObjects` will be discussed in Section 6.)

¹ The `getTime` method of our editor caches the time it returns. If you call it twice within the same millisecond, it will add an extra millisecond in order to avoid the same time stamp on multiple commands, cf. [1]

```

1 package JavaPackages;
2 ...
3 public class JavaPackagesEditor {
4     private Map<String, Object> mapOfModelObjects
5         = new LinkedHashMap<>();
6     private Map<String, Object> mapOfFrames
7         = new LinkedHashMap<>();
8     private Map<String, Object> mapOfParsedObjects
9         = new LinkedHashMap<>();
10    ...
11 }

```



Listing 1.15. Maps for model objects and object frames

When we need an object as context (black rule part) we call method `getObjectFrame` on the corresponding `editor`. Lines 7 to 10 of Listing 1.16 will be discussed in Section 6. Line 11 of Listing 1.16 first tries to retrieve the desired object from the `mapOfModelObjects`. If that fails, Line 15 tries to retrieve the desired object from the `mapOfFrames`. If this still fails, Line 19 to 24 first use reflection in order to create the desired object and to initialize its `id` and then the object is added to the `mapOfFrames` and returned.

```

1 package JavaPackages;
2 ...
3 public class JavaPackagesEditor {
4     ...
5     public Object getObjectFrame(Class clazz, String id) {
6         try {
7             Object modelObject = mapOfParsedObjects.get(id);
8             if (modelObject != null) {
9                 return modelObject;
10            }
11            modelObject = mapOfModelObjects.get(id);
12            if (modelObject != null) {
13                return modelObject;
14            }
15            modelObject = mapOfFrames.get(id);
16            if (modelObject != null) {
17                return modelObject;
18            }
19            modelObject = clazz.getConstructor().newInstance();
20            Method setIdMethod
21                = clazz.getMethod("setId", String.class);
22            setIdMethod.invoke(modelObject, id);
23            mapOfFrames.put(id, modelObject);
24            return modelObject;
25        } catch (Exception e) {

```

```

26         throw new RuntimeException(e);
27     }
28 }
29 ...
30 }

```

Listing 1.16. getObjectFrame method

Similarly, when we want to create a model object explicitly (green rule parts) we call method `getOrCreate` on the corresponding editor. Again, Lines 6 to 10 will be discussed in Section 6. Line 11 of Listing 1.17 tries to retrieve the desired model object from our `mapOfModelObjects`. If that fails, Line 15 calls method `getObjectFrame` which either retrieves the desired object or creates it. Then Lines 16 and 17 promote the desired object into the `mapOfModelObjects` and Line 18 returns it.

```

1 package JavaPackages;
2 ...
3 public class JavaPackagesEditor {
4     ...
5     public Object getOrCreate(Class clazz, String id) {
6         Object modelObject = mapOfParsedObjects.get(id);
7         if (modelObject != null) {
8             mapOfModelObjects.put(id, modelObject);
9             return modelObject;
10        }
11        modelObject = mapOfModelObjects.get(id);
12        if (modelObject != null) {
13            return modelObject;
14        }
15        modelObject = getObjectFrame(clazz, id);
16        mapOfFrames.remove(id);
17        mapOfModelObjects.put(id, modelObject);
18        return modelObject;
19    }
20    ...
21 }

```

Listing 1.17. getOrCreate method

In order to remove an object from our model we call method `removeModelObject` on the corresponding editor. Line 6 of Listing 1.18 tries to remove the object from our `mapOfModelObjects`. If that succeeds, Line 8 adds the removed object to our `mapOfFrames` (as it may still be used as context by some other command).

```

1 package JavaPackages;
2 ...
3 public class JavaPackagesEditor {
4     ...

```

```

5      public Object removeModelObject(String id) {
6          Object oldObject = mapOfModelObjects.remove(id);
7          if (oldObject != null) {
8              mapOfFrames.put(id, oldObject);
9          }
10         return mapOfFrames.get(id);
11     }
12     ...
13 }

```

Listing 1.18. removeModelObject method

Actually, to remove some object from our model, we need to (implement and) call the `undo` method of the responsible command in order to remove the complete increment and to leave our model in a consistent state. In addition, we have to remove the corresponding command from our `activeCommands` hash table. And we have to inform all other tools during subsequent model synchronisation. In addition, we have to be careful, in order to prevent the re-execution of the removed command when we receive it again (e.g. from another tool). Our implementation uses a special `RemoveCommand` to achieve this, cf. Listing 1.19.

```

1 package JavaPackages;
2 ...
3 public class RemoveCommand extends ModelCommand {
4     public Object run(JavaPackagesEditor editor) {
5         editor.removeModelObject(getId());
6         ModelCommand oldCommand
7             = editor.getActiveCommands().get(getId());
8         if (oldCommand != null) {
9             oldCommand.undo(editor);
10        }
11        return null;
12    }

```

Listing 1.19. Command execution

We may e.g. call (`RemoveCommand c`) on our `editor` and the `editor` may already have a (`HaveLeaf c sub 1.1`) command in its set of `activeCommands`. Then the time stamp compare within the `execute` method of our `editor` will find that the `RemoveCommand` is later than the `HaveLeaf` command and it will call `run` on the `RemoveCommand`. Line 5 of Listing 1.19 assumes per default that the command to be removed has created at least one model object with the same `id` as the command (and the `RemoveCommand`). Thus, Line 5 calls `removeModelObject` with that `id`. Therefore, simple commands that create and initialize only a single object do not even have to implement the `undo` method as the `RemoveCommand` already does this job. If you do not want to rely on this default assumption, you may easily omit this line in your implementation.

Line 7 of Listing 1.19 retrieves the command to be removed from our `activeCommands` and Line 9 calls its `undo` method. Afterwards, the `execute` method of our `editor`

replaces the `undo` command with the `RemoveCommand` within our `activeCommands`, cf. Line 28 of Listing 1.14. On model synchronization the `RemoveCommand` will be send to the other tool(s) and perform the same operation there.

Note, we need to keep the `RemoveCommand` in our `activeCommands` table until we are sure that all other tools (and all persistent copies of our `activeCommands` table have overwritten their copy of e.g. the `HaveLeaf` command. If we remove the `RemoveCommand` too early and if we receive the overwritten `HaveLeaf` command thereafter, we would not notice that the `HaveLeaf` command has been removed but we would re-execute it.



6 Parsing

Sometimes, you may want to edit your model directly, cf. Line 12 to 23 of Listing 1.20. Then you need to parse your (modified) model in order to retrieve the set of commands that correspond to it and to synchronize your changes with other models. Parsing basically requires to split your model into increments where each increment corresponds to a certain command. In our approach, we follow the convention that each command creates one model object that gets the same `id` as the command. This unique model object becomes the nucleus for each increment. Thus, in our approach the identification of model increments starts with these nucleus objects and parsing just needs to collect the remaining parts of the increment.



```

1 package JavaPackages;
2 ...
3 public class TestPackageToDoc
4         implements PropertyChangeListener {
5     ...
6     @Test
7     public void testManualChangesAndParsing() {
8         JavaPackagesEditor javaPackagesEditor
9             = new JavaPackagesEditor();
10        startSituation(javaPackagesEditor);
11        registerModelObjectListener(javaPackagesEditor, this);
12        JavaPackage nRoot = new JavaPackage().setId("nRoot");
13        JavaPackage root = (JavaPackage)
14            javaPackagesEditor.getModelObject("root");
15        nRoot.withSubPackages(root);
16        JavaPackage sub = (JavaPackage)
17            javaPackagesEditor.getModelObject("sub");
18        sub.setUp(null);
19        JavaClass c2 = new JavaClass()
20            .setUp(sub).setVTag("1.1").setId("c2");
21        JavaClass c = (JavaClass)
22            javaPackagesEditor.getModelObject("c");
23        c.setVTag("1.1");

```

```

24      Set allObjects = Yaml.findAllObjects(nRoot, sub);
25      javaPackagesEditor.parse(changedObjects);
26      ...
27  }
28  ...
29  }

```

Listing 1.20. Command execution

In [9] the parsing of a *HaveLeaf* rule requires that the up-per **JavaPackage** has already been parsed either by a **HaveRoot** or by an **HaveSubUnit** rule. Similarly, the parsing of a **HaveSubUnit** rule requires that the up-per **JavaPackage** for the corresponding sub **JavaPackage** has already been parsed. Thus, in [9] parsing must start with the **HaveRoot** rule and then you parse the sub *JavaPackages* of these root(s) and then the sub-sub packages until you reach the *JavaClass* leaves. These parsing dependencies between general Triple Graph Grammar rules makes the parsing of Triple Graph Grammars (and especially incremental parsing) very complex.

Compared to general Triple Graph Grammars, parsing of Commutative Event Sourcing models is a piece of cake. As we have no rule dependencies, even incremental parsing becomes easy. Therefore, Line 11 of Listing 1.20 subscribes a property change listener to our object model. (See [1] for implementation details.) This property change listener collects all objects affected by the changes executed in Lines 12 to 23. This allows us to call the **parse** method of our **editor** with just the set of **changedObjects** in Line 25. Alternatively, Line 24 uses our object serialization mechanism to collect all objects that belong to the modified model and we could use the set of **allObjects** within our **parse** call.

Line 6 of Listing 1.21 registers all objects that shall be parsed into our **mapOfParsedObjects** (cf. Listing 1.15, see [1] for details). For each object that shall be parsed, Line 10 calls method **findCommands**, which does the actual parsing. Method **findCommands** retrieves a set of command **prototypes** (Line 26 of Listing 1.21). Then Line 28 calls the **parse** method of each command prototype. These parse methods analyse the current object, whether it is the nucleus of some increment that matches that command, cf. Listing 1.12 and Listing 1.13. If the command fits, its **parse** method returns a new copy of the corresponding command with all command parameters assigned, properly. On success, Line 30 of Listing 1.21 collects the parsed command in the set of **allCommands**. Once we have identified all commands that correspond to the objects to be parsed, we look for **oldCommands** that will be overwritten by the execution of a new command, cf. Line 14 to 16 of Listing 1.21. If there is already an old command with the same parameters, we do not overwrite it in order to keep the old time stamp. If the new command is actually different (or there is no old command) we execute it (Line 17) in order to update our set of **activeCommands**. (Note, executing a command that has been parsed will edit the corresponding increment but this edit should just re-assign the values that have been used during parsing. Thus, if the **run** and the **parse** method work consistently, running the command does no harm (but may also be skipped).)

```

1  package JavaPackages;
2  ...
3  public class JavaPackagesEditor {
4  ...
5  public void parse(Collection allObjects) {
6      registerParsedObjects(allObjects);
7      ArrayList<ModelCommand> allCommandsFromParsing
8          = new ArrayList<>();
9      for (Object object : allObjects) {
10         findCommands(allCommandsFromParsing, object);
11     }
12     for (ModelCommand command : allCommandsFromParsing) {
13         String id = command.getId();
14         ModelCommand oldCommand = activeCommands.get(id);
15         if (oldCommand == null ||
16             ! equalsButTime(oldCommand, command)) {
17             execute(command);
18         }
19     }
20 }
21
22 private ModelCommand findCommands(
23     ArrayList<ModelCommand> allCommands,
24     Object object) {
25     ArrayList<ModelCommand> prototypes
26         = haveCommandPrototypes();
27     for (ModelCommand prototype : prototypes) {
28         ModelCommand command = prototype.parse(object);
29         if (command != null) {
30             allCommands.add(command);
31         }
32     }
33     return null;
34 }
35 ...
36 }

```

Listing 1.21. Command execution

There is still one little design issue to be discussed. In our example Line 12 of Listing 1.20 creates the `JavaPackage nRoot`, directly. Our change listener will collect the new `nRoot` object as soon as it is linked to the old model object `root` (Line 15). Thus, parsing will create a (`HaveRoot nRoot`) command. When we execute this (`HaveRoot nRoot`) command, its `run` method calls `getOrCreate` (Line 6 of Listing 1.1) to retrieve the desired model object. Now, some parts of our (testing) program may still hold a reference to the directly created `nRoot` object. Thus we want `getOrCreate` to retrieve this directly created `nRoot` object

and `getOrCreate` should not create a new model object. To achieve this, Line 6 of Listing 1.21 registers all directly changed objects in our `mapOfParsedObjects` and method `getOrCreate` tries to retrieve the required object from there (Lines 6 to 10 of Listing 1.17). Method `getObjectFrame` works similarly (Listing 1.16).

7 Conclusions

To revisit the title of this paper, Commutative Event Sourcing may be considered as just a restricted variant of Triple Graph Grammars where the commands or rules are either overwriting or commutative. This frees Commutative Event Sourcing from handling dependencies between commands / rules. Thereby, model synchronisation, collaborative editing, and even incremental parsing is facilitated, considerably.

This paper tries to provide sufficient details such that you may implement Commutative Event Sourcing for your tool(s), manually. You may adapt our concepts without relying on a special programming language, library, framework, or tool. You may also copy large parts of our example implementation from [1]. Actually, our editors are quite generic, only the set of command prototypes is model specific. [1] also provides a code generator for editors and the generic parts of commands, if you want to use that. [1] even provides a simple interpreter for TGG like Commutative Event Sourcing patterns. However, using this rule interpreter requires a certain learning curve for writing these patterns and pattern execution is hard to debug. Thus, beginners may be better off by implementing their commands, manually.

Commutative event sourcing requires that commands / rules are either overwriting or commutative. This is quite a restriction compared to general Triple Graph Grammars. But it facilitates implementation. We like to compare this with the introduction of LALR(k) grammars [7] in compiler construction that reduced memory consumption within compilers compared to more general LR(k) grammars. LALR(k) grammars reduce the set of parseable languages, slightly, but acceptable.

Our approach also makes extensive use of `ids` for (cross model) object reference. If you do not want `id` attributes in your model, your editor may use an additional `objectToId` map in order to store and retrieve object `ids`. We did so e.g. when using Commutative Event Sourcing for the solution of a model synchronisation case in the Transformation Tool Contest 2020 [5] and [1]. If you use `ids` for cross model referencing, you want to assign these `ids` directly and you do not want e.g. a database system generating your `ids`. Directly edited `ids` are usually considered an anti pattern. And yes, this is a design challenge for Commutative Event Sourcing.

Commutative Event Sourcing considers two models to be equivalent if they contain the same objects with the same `ids` and same attribute values and the same *set* of links. In case of a to-many association we handle the neighbors as a set and not as a list, i.e. we consider the order of the neighbors as not relevant. If your commands are commutative and show up in any order it is just



hard to achieve a certain order in a list. This again is a design challenge, e.g. if you want to show a number of objects in the graphical user interface on two different tools and you want that both users see same same order. (Well just sort them.) Neglecting the order of lists is also a challenge for models with a textual representation: you may not want an arbitrary order of your program statements (and sorting want help).

However, we have used Commutative Event Sourcing with great success in [5] and with some other model synchronisation problems e.g. for BPMN diagrams and a textual Workflow language [1]. We also used it in our `MicroServices` course in Winter Term 2019 / 2020 [2]. According to our experiences, the Commutative Event Sourcing approach to model synchronisation is quite easy to engineer and works quite reliable.

References

1. Github example implementation. <https://github.com/fujaba/fulibServiceGenerator>, <https://github.com/fujaba/fulibServiceGenerator/blob/master/test/src/test/java/javaPackagesToJavaDoc/TestPackageToDoc.java>, last viewed 31.08.2020
2. Youtube playlist labor microservices, winter term 2019 / 2020. <https://www.youtube.com/playlist?list=PLohPa1TMsvQrI0FaMTySbGr02JkQs5MhQ>, last viewed 31.08.2020
3. Anjorin, A., Diskin, Z., Jouault, F., Ko, H.S., Leblebici, E., Westfechtel, B.: Benchmarkx reloaded: A practical benchmark framework for bidirectional transformations. In: *BX@ ETAPS*. pp. 15–30 (2017)
4. Copei, S., Sälzer, M., Zündorf, A.: Multidirectional transformations for microservices. In: *Proc. Dagstuhl Seminar*. vol. 18491 (2019)
5. Copei, S., Zündorf, A.: The fulib solution to the ttc 2020 migration case
6. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional transformations: A cross-discipline perspective. In: *International Conference on Theory and Practice of Model Transformations*. pp. 260–283. Springer (2009)
7. DeRemer, F., Pennello, T.: Efficient computation of lalr (1) look-ahead sets. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **4**(4), 615–649 (1982)
8. Evans, E.: *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional (2004)
9. Fritsche, L., Kosiol, J., Schürr, A., Taentzer, G.: Avoiding unnecessary information loss: Correct and efficient model synchronization based on triple graph grammars. *arXiv preprint arXiv:2005.14510* (2020)
10. Hildebrandt, S., Lambers, L., Giese, H., Rieke, J., Greenyer, J., Schäfer, W., Lauder, M., Anjorin, A., Schürr, A.: A survey of triple graph grammar tools. *Electronic Communications of the EASST* **57** (2013)
11. Leblebici, E., Anjorin, A., Schürr, A., Hildebrandt, S., Rieke, J., Greenyer, J.: A comparison of incremental triple graph grammar tools. *Electronic Communications of the EASST* **67** (2014)
12. Rozenberg, G.: *Handbook of graph grammars and computing by graph transformation*, vol. 1. World scientific (1997)

13. Schneider, C., Zündorf, A., Niere, J.: Coobra-a small step for development tools to collaborative environments. In: Workshop on Directions in Software Engineering Environments. Citeseer (2004)
14. Schürr, A.: Specification of graph translators with triple graph grammars. In: International Workshop on Graph-Theoretic Concepts in Computer Science. pp. 151–163. Springer (1994)
15. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
16. Stevens, P.: Bidirectional model transformations in qvt: semantic issues and open questions. *Software & Systems Modeling* **9**(1), 7 (2010)
17. Vernon, V.: Implementing domain-driven design. Addison-Wesley (2013)