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 manually. You are not restricted to Java as your programming language and you do not need to use a proprietary library, framework, or tool. You do not even have to dig into graph grammar theory.

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 and to collaborative editing.

In the area of bidirectional (BX) transformations there are various approaches attacking the problem of model synchronisation cf. [7,4]. Among the various approaches, we conside Triple Graph Grammars (TGGs) [15] to be the most mature and most practical solution with a lot of tool support [11]. Recent development in TGG tools provide support for incremental model synchronisation [12], i.e. the effort for model synchronisation is proportional to the model change performed. In [10] 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 [16]) 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 discuss, CES is a restricted form of a TGG 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 [10]. The running example of [10] 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.

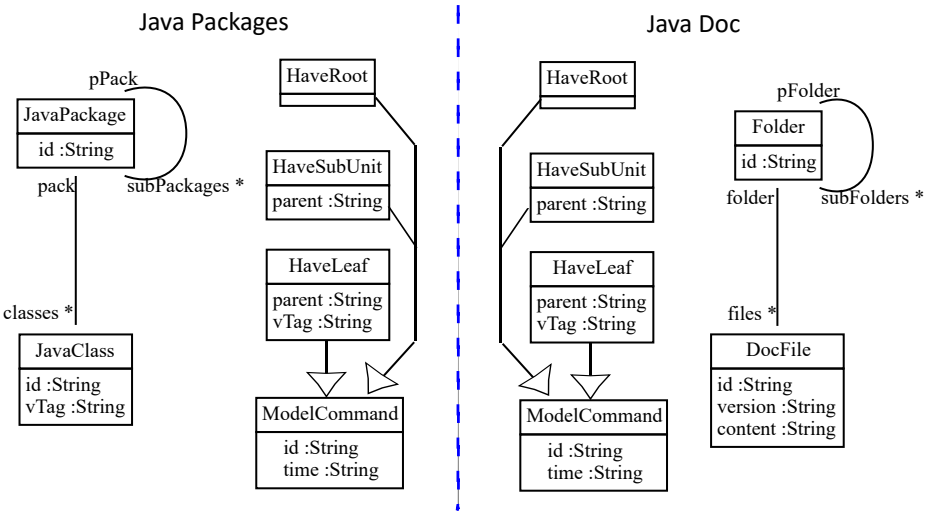

Fig. 1. Classes for Java and JavaDoc structures plus edit commands

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 [10] 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 [10]. 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 [10] 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`, `remove`, 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 pp` attached to `p` via an `pPack` 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, this.getId());
7          p.setPPack(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 6. This hash table is e.g. used in the `HaveSubUnit` command to look up the required `pp JavaPackage`, cf. method `getObjectFrame` in Line 8 of Listing 1.2 and Section 6. Note, in the left subrule of the `HaveSubUnit` rule of Figure 2 the upper `JavaPackage pp` is shown in black color. This means the execution of the subrule requires that `pp` exists as context for the creation of the sub package `p`. The green `pPack` link requires that `p` needs to be connected to `pp` via an `pPack` link (which simultaneously creates the `subPackages` link in the reverse direction).

```
1  package JavaPackages;
2  public class HaveSubUnit extends ModelCommand {
```

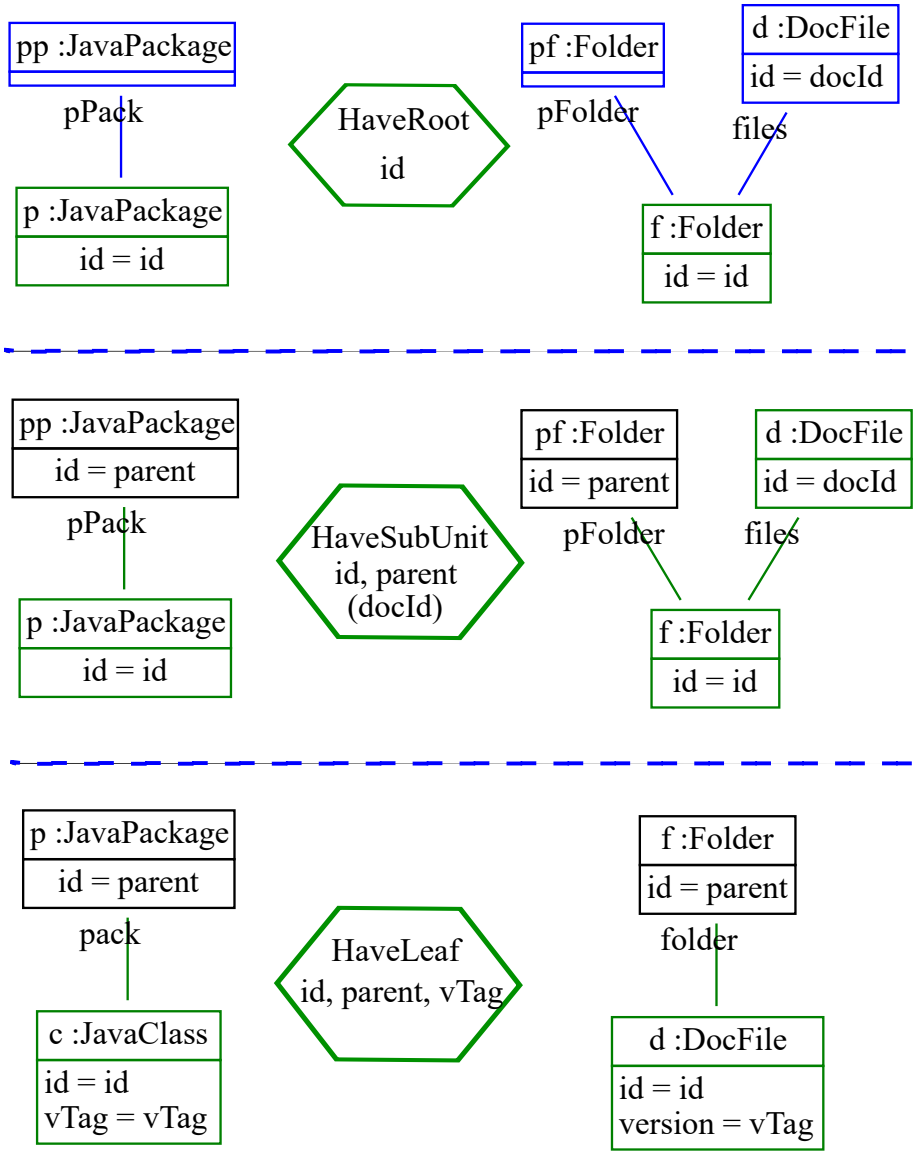

Fig. 2. Triple Graph Grammar (like) rules for Java and JavaDoc structures

```
3     @Override
4     public Object run(JavaPackagesEditor editor) {
5        JavaPackage p = (JavaPackage) editor
6              .getOrCreate(JavaPackage.class, this.getId());
7        JavaPackage pp = (JavaPackage) editor
8              .getObjectFrame(JavaPackage.class, this.parent);
9        p.setPPack(pp);
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, this.getId());
8        JavaPackage p = (JavaPackage) editor
9              .getObjectFrame(JavaPackage.class, this.parent);
10       c.setPack(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 {
4     private void startSituation(JavaPackagesEditor editor) {
5        ModelCommand cmd = new HaveRoot().setId("org");
6        editor.execute(cmd);
7        cmd = new HaveSubUnit().setParent("org").setId("fulib");
8        editor.execute(cmd);
9        cmd = new HaveSubUnit().setParent("fulib").setId("serv");
10       editor.execute(cmd);
11       cmd = new HaveLeaf().setParent("serv")
```

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

Listing 1.4. Invoking triple rules or commands

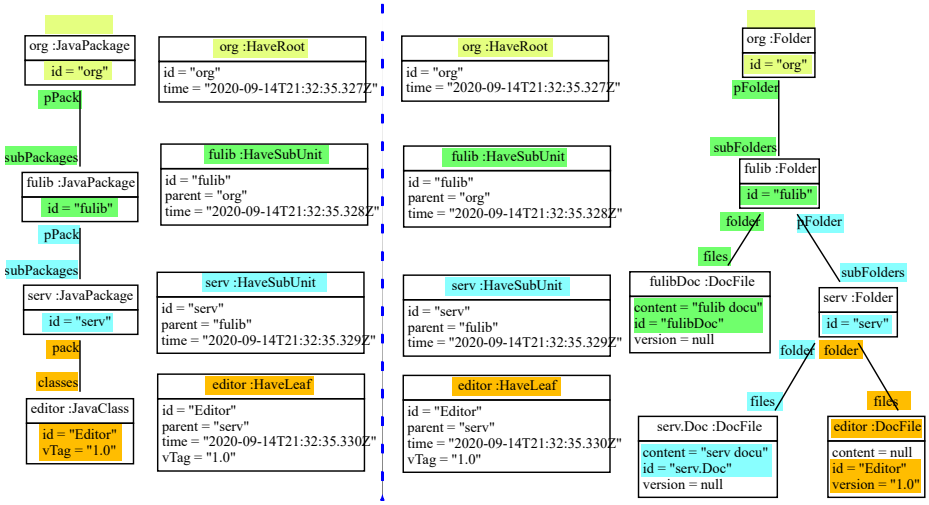

Fig. 3. Objects for Java and JavaDoc structures plus commands (colors cf. Section 4)

Listing 1.5 shows our manual implementation of the `HaveRoot` command for `JavaDoc` structures. This implements the right subrule 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 after rule execution such a `DocFile` must not (no longer) 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, this.getId());
7          f.setPFolder(null);
8          String docId = this.getId() + ".Doc";
9          DocFile d = f.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, this.getId());
7          Folder pf = (Folder) editor.
8                  getObjectFrame(Folder.class, parent);
9          f.setPFolder(pf);
10         String docId = this.getId() + ".Doc";
11         DocFile d = (DocFile) editor
12                 .getOrCreate(DocFile.class, docId);
13         d.setContent(this.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, this.getId());
7          Folder f = (Folder) editor
8                  .getObjectFrame(Folder.class, parent);
9          d.setFolder(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 of 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 {
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 `remove` and `parse` functionality. Removing the effects of 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 `remove` 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 `remove` 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 remove(JavaPackagesEditor editor)
6      {
7          editor.removeModelObject(this.getId());
```

```
8       }
9       ...
```

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

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

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

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

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

Usually, TGG rule applications depend on each other. For example if we call `remove` on the `JavaPackages HaveRoot` command of our example, this would remove the `org JavaPackage` from our model, cf. Figure 3. This would leave the `fulib JavaPackage` without a parent. Thus the `HaveSubUnit` TGG rule does no longer match for the `fulib` object. This is important as on model synchronisation the right sub rule of `HaveSubUnit` creates the `fulibDoc DocFile` which is no longer valid. To repair this, a standard TGG approach has to `remove` dependant rule applications whenever their context becomes invalid. Thus on `remove` of the `HaveRoot` command, a standard repair step would also `remove` the `HaveSubUnit` command for the `fulib JavaPackage` and in turn `remove` the commands for the `serv` and for the `editor` objects. Via model synchronisation all `JavaDoc` objects will be removed, either. To keep the lower model parts, one would apply a `HaveRoot` command on `fulib` and rerun the `HaveSubUnit` and `HaveLeaf` commands on `serv` and `editor`. Formally, the whole model would be deleted and reconstructed. In [10] this is called a cascading delete and [10] 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 `fulib` 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` rule. In a manual implementation you have to spot the overlap of the `HaveRoot` and the `HaveSubUnit` rules yourself and you have to design theses 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. [10] 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 tries to find applications of rules that use the "root" rules in their context. In turn, you apply rules where the context has become available. In our example this results in some kind of top-down parsing starting with the `org JavaPackage` and descending to `subPackages` and `classes`, recursively. In general, TGG parsing may be even more complicated, cf. [15]. Again Commutative Event Sourcing allows us to facilitate parsing considerably as we will discuss in Section 7.

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 7. 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 `pPack`. If the current package has no sub packages and contains no `JavaClass`es, 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 `RemoveCommand`s 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.getPPack() != 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.getPPack() == null) {
11          return null;
12       }
13       ModelCommand modelCommand = new HaveSubUnit()
14             .setParent(currentPackage.getPPack().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 [9] and [18] as a means for communication between multiple domains or (micro) services. Event Sourcing may also be used as mechanism for model persistence. 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 org`) event and a (`HaveSubUnit fulib org`) event and for some reasons you receive only the latter, cf. Figure 4. In [14] we deal with this problem by adding `time` stamps to all events / commands and by extending each event with the *time* stamp of its predecessor. Thus if you receive event (`HaveSubUnit fulib org 13:02 13:01`) which has been raised at 13:02 and which has a predecessor command that has been raised at 13:01 and you have not yet received the 13:01 command, then you postpone the execution of the `13:02` event and ask for resubmission of the `13:01` event, cf. Figure 4. Time stamps also allow to detect duplicated receipt of the same event. In addition, if there are two independent tools (e.g. the editors of Alice and Bob) that raise event (`HaveLeaf Editor fulib 1.0 13:10 13:02`) and (`HaveLeaf Editor fulib 1.1 13:11 13:02`) you have a merge conflict and the time stamps allow you to resolve such merge conflicts, deterministically.

While time stamps are a good idea, in [14] postponing command execution until all predecessor commands have arrived was quite tricky: you have to detect the missing of a predecessor, you have to ask for resubmission you have to wait for the predecessor to arrive, there may be a cascade of pre-predecessors you also have to ask and to wait for, finally you apply the commands in their correct order. Well, unless there is collaborative editing of multiple editors: with collaborative editing, when you receive a command with time stamp 13:30 (and predecessor 13:11) and you already got the 13:11 event, there still might be e.g. an event with time stamp 13:23 (and also predecessor 13:11) that just did not reach you, yet. Thus, to execute collaborative commands in a correct timely order opens a new can of worms.

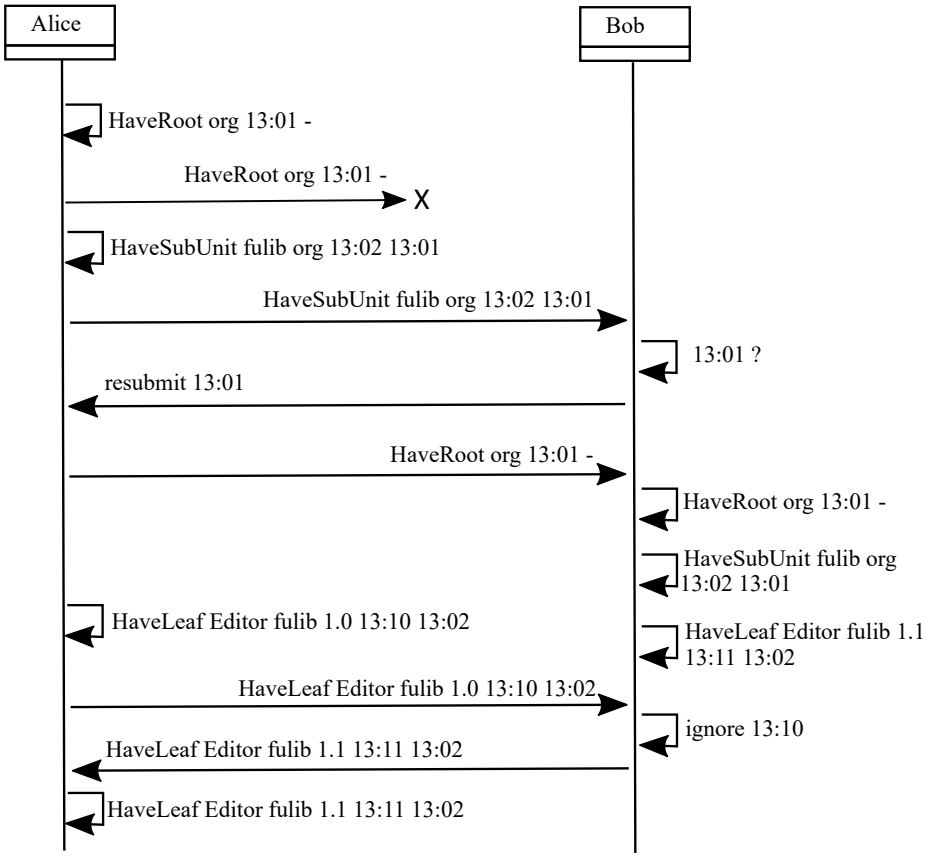

Fig. 4. Collaborative Editing

To overcome the problem of rule ordering, we want our commands to be executed in any order, i.e. to become commutative. Thus, we want to get rid of TGG rule dependencies. One reason for dependencies between TGG rule applications is that rule execution requires that some context (black) parts must already exist in order to connect new elements to their context. For example, a (`HaveLeaf Editor serv`) command for `JavaPackages` needs to connect the new `Editor JavaClass` to an (existing) `serv JavaPackage` via a `pack - classes` link, cf. Figure 2 and Listing 1.3. In our approach we are able to execute the `HaveLeaf Editor serv` command before the `serv JavaPackage` is created. We achieve this by using `id`s 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 9 of Listing 1.3. The details of the `getOrCreate` and `getObjectFrame` mechanism are explained in Section 7. Thus, our (`HaveLeaf Editor serv`) command creates a context object with `id serv` and creates the *pack* link to it. When we execute

the (`HaveSubUnit serv fulib`) command later, it uses `getOrCreate(serv)` to retrieve the `serv JavaPackage` that has already been created by our `HaveLeaf` command and to connect the `serv JavaPackage` to its `pPack fulib`. The `fulib JavaPackage` is again retrieved (or created) via `getObjectFrame`.

This `getOrCreate` mechanism is inspired by QVT Relations [2] and [13], however QVT Relations allows to combine any attributes that may be used as keys while we restrict this to just `id`s, in order to facilitate a manual implementation. In addition, QVT Relations still deploys a hierarchy of rules which requires considerable implementation effort for the handling of rule dependencies.

Due to our `getOrCreate`, mechanism together with some additional rule restrictions that are discussed in Section 4, in our approach it is possible to execute commands in any order, i.e. our commands are commutative. Having commutative commands gets rid of the effort for command ordering. And it facilitates parsing, cf. Section 7.

There is one additional problem with general event sourcing: usually your event history grows over time. For example if you have a (`HaveLeaf Editor serv 1.1`) command that creates an `Editor` object and assigns `1.1` to its `vTag` and than you have a new (`HaveLeaf Editor serv 1.2`) command that just changes the `vTag` to `1.2` and later on you change `vTag` again and again. Now your event history may contain a large number of `HaveLeaf` events that only differ in their `vTag` parameter. Actually, it would suffice to keep only the latest `HaveLeaf` command in order to recreate the final model or to synchronize with some other model. In our approach we want a simple mechanism to identify events that overwrite each other in order to be able to restrict the size of our event history to be proportional to the size of our model. Therefore, our implementation uses command `id`s and in our implementation two commands that have the same `id` parameter must overwrite each other's effects such that it suffices to keep one event per command `id` in our history, cf. Section 4 and Listing 1.14.

Next, we provide some theory that specifies the requirements for Commutative Event Sourcing and compressed event histories, formally. 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 [5]. 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 [17] 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 $e = (t, i, x_1, ..., x_n)$ 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 $\overline{e} = (e_1, ..., e_n)$ and denote by $\overline{E}$ the set of all possible event series with events from E. Similarly we use sets of events $\tilde{e} = \{e_1, ..., e_n\} \in \mathcal{P}(E)$.

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 applica-

tion of an event series $\overline{e} = (e_1, \ldots, e_n)$ to a model m as
$apply(\overline{e}, m) = apply(e_n, \ldots, apply(e_2, apply(e_1, m)) \ldots)$.
Similarly, the application of a set of events $\tilde{e}$ is defined as the applications of all its elements in some order.

Now, we want a simple mechanism that allows us to restrict the size of our command histories. Therefore, Definition 2 requires that events with the same `id` overwrite each others effects, i.e. one renders the other ineffective. 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 $\overline{e}$ at position p iff $apply(\overline{e} \setminus e_p, \emptyset) = apply(\overline{e}, \emptyset)$.*
We call e_p effective, otherwise.
We call an event series $\overline{e}$ effective iff it contains only effective events.

Definition 2. *(overwriting) We call events $e_1 = (t_1, i_1, ...)$ and $e_2 = (t_2, i_2, ...)$ with identifier $i_1 = i_2$ overwriting, if for all event series $\overline{e}$ that contain e_1 and e_2 holds either e_1 or e_2 is ineffective.*

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 (unordered) 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).

Definition 4. *(active event sets $\tilde{e}$)*
For any event series $\overline{e}$ where for all e_1, e_2 in $\overline{e}$ holds
(1) e_1 and e_2 are commutative and
(2) if e_1 and e_2 have the same identifier, they are overwriting,
we define the set of active events $\tilde{e} = \{\ e \text{ in } \overline{e} \mid e \text{ is not overwritten}\ \}$.

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

Definition 5. *(M_{CES}) A model $m = apply(\overline{e}, \emptyset)$ is supporting Commutative Event Sourcing if and only if*
all events $e_1 = (t_1, i_1, ...)$ and $e_2 = (t_2, i_2, ...)$ in $\overline{e}$ are commutative and overwriting, (if $i_1 = i_2$).
Thus, there exists an active event set $\tilde{e}$ with $apply(\tilde{e}, \emptyset) = m$.
In addition we require a function $parse : M \to \mathcal{P}(E)$ such that $parse(m) = \tilde{e}$.
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 active 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 active 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. We will discuss this in our conclusions.

4 Achieving Commutative Event Sourcing

This section provides some simple design rules to achieve Commutative Event Sourcing. Our 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. With respect to our `getOrCreate` mechanism, objects are part of an increment if their `id` attribute is a part of the increment. In Figure 3 we have color coded different increments and their commands.

To achieve commutativity for our commands, we first look on commutativity for TGG rules. Basically, a TGG rule has a context (black) part that shall already exist and that is not modified and a main (green and blue) part that defines its increment, i.e. the attributes and links that will be edited on rule execution. In general, a TGG rule may have an arbitrary complex context (black) part. You may require multiple objects that shall be connected by various links and you may have additional attribute constraints and application conditions. However, such complex context parts potentially create rule dependencies: for example, if a rule requires some link within its context part and later this link is removed, the rule application is no longer valid and it must be removed, too. This may cause cascading deletes. To avoid such rule dependencies, we restrict the context (black) part of our rules to simple objects with an `id` attribute constraint. No links between context objects and no general attribute constraints.

The main (green and blue) parts of a rule result in active changes of attribute values and links. This may interfere with another rule, if the other rule changes the same attributes or links, differently. In that case rule application order would matter. Thus, we require that the increment edited by the main rule parts must not overlap with the increment of other rule applications, i.e. no two rule applications shall edit the same attribute or link.

Together, simple contexts and not overlapping main parts achieve commutativity. We really would like to prove these criteria to be sufficient for commutativity, however we would need to extend our section on theory in order to formalize the set of attributes and links edited by a rule application. This needs

some graph grammar theory and this paper promises to achieve model synchronisation without the need to go into graph grammar theory. If you implement the commands manually, the graph grammar theory results will not apply for you anyway. Thus, to validate the commutativity of your TGG rules, you will have to execute them in different order and to check whether all orders achieve the same result.

During the manual implementation of our commands, it is straight forward to restrict ourselves to simple context (black) objects: either do not check for attribute values or links or if you do check an attribute, consider it as a part of your increment, i.e. you edit it.

To achieve non-overlapping increments we first introduce a convention: each command shall have exactly one *core* object that has the same `id` as the command itself. Based on this convention, to find non-overlapping increments we start with some reasonably representative object model example. Then we choose some object as the core object of our first command and we mark the `id` attribute of this object with a certain color. In Figure 3 we may e.g. start with the `Editor` object in the lower left corner and mark its `id` attribute with orange. Next, we look for other attributes of the same object that our command may initialize in the same step and mark them with the same color. In our example, we choose the `vTag` attribute of our `Editor` object. Next, we look for to-one links that connect our core object to other objects. In our example we choose the `pack - classes` link attached to our `Editor` object and mark it with orange, too. Now we look for other `core` objects of the same type and tries to mark a similar increment induced by this new `core` object. In Figure 3 there is no other `JavaClass` object, thus we go on with a new core object of a different type, e.g. with the `serv` object of type `JavaPackage`. The `serv` object has no other attributes but a `pPack - subPackages` link of cardinality to-one. We color the `id` attribute and the link in blue. Now we look at the `fulib` object on the left of Figure 3 which is also of type `JavaPackage`. Here we use green color to mark a similar increment. If we look at the `org` object there is no `pPack - subPackages` link attached to it. However, there is an empty field within the `org` object that could hold a `pPack` link. To make the `org` increment similar to the other JavaPackage increments, we mark the empty space above the `org` object with yellow color. Now we have colored all attributes and all links of our `JavaPackages` example model and no attribute and no link is marked by two colors. Thus, we have identified non-overlapping increments that cover all elements of our example model.

We are now ready to define our commands: for each core object we create a command object with a certain type. For each attribute of an increment, the command object gets an appropriate parameter attribute. For each link of our increment the command gets a parameter attribute that holds the `id` of the target object. Usually, we use one command type for each increment type or for each core object type. If there are two core objects of the same type which have increments that differ in their structure, you might use two different command types or one command type which distinguishes the two cases, internally. In our example, we use command type `HaveRoot` for the `org` object and command type

HaveSubUnit for the fulib and the serv object. These two commands differ in the parent parameter and in their handling of the pPack link: the former deletes the pPack link and the latter creates it. Generally, increments (with core objects) of the same type must be similar, i.e. they edit similar sets of attributes and links. Accordingly, commands of different types that edit increments of the same type must edit similar sets of attributes and links. Therefore, the HaveRoot command has to assign a defined value (i.e. null) to the pPack field of its core object. Actually, in a manual implementation we would probably use only one command type for JavaPackage objects, as it is easy to handle both cases in one implementation. However, the example stems from [10] and [10] uses TGG rules and with TGGs you need different rules for different cases. Thus, we use two command types to facilitate the comparison.

Note, as all commands that edit the same increment type or core objects with the same type edit similar sets of attributes and links our commands are overwriting by construction: if two commands have the same id, they edit the same core object and the same increment and as discussed all parts of the increment get a well defined value. Thus, one command will overwrite all attributes and links that have been edited by the other command and thus it suffices to keep only the last command in our command history.

Let's now have a look at the JavaDoc example model at the right side of Figure 3. Given the increments of the left model, we now have to identify the corresponding increments on the right model. For the orange increment on the bottom, this is the Editor DocFile with its id attribute, its version attribute and its folder - files link. In Figure 3 this is easily spotted by comparing object ids and attribute values and link targets and names. Usually, you have only one example model as a start and you construct the target model "incrementally". Thus, you identify which objects, attributes and links need to be added to the target model in order to represent the information that is provided by the source increment or by the corresponding command and its parameters. And you reuse the ids of the source model within the target model in order to establish the desired correspondences.

Note, the content attribute of our Editor object on the right is not marked with orange. Actually, the content attribute of a DocFile has no correspondence within the JavaPackages model and thus it will not be addressed by model synchronisation but we will introduce a separate HaveContent command within the JavaDoc model later on, cf. [1].

For the HaveSubUnit increments or commands the situation is somewhat more complicated: Within the JavaDoc model, a sub package is represented by a Folder object and by a special DocFile that describes the sub package. If we look e.g. at the blue serv increment in Figure 3, within the JavaDoc model this increment is represented by the serv object (of type Folder) and by the serv.Doc object (of type DocFile) and by its content attribute and the attached folder - files and pFolder - subFolders links. Thus, our increment contains two model objects, the core object with id serv and a dependent object with id serv.Doc. In general, a Folder object may contain multiple DocFile objects.

In order to identify the `DocFile` object that describes the `Folder` itself, our example uses the convention that the describing `DocFile` has an `id` that is equal to the `id` of its `Folder` plus a `".Doc"` suffix, cf. Figure 3 and Listing 1.6. With this convention it is still possible to identify all parts of an increment by starting at some model object and collecting the attached parts. If you start e.g. with the `serv.Doc` object you may identify the corresponding `serv` object either using our naming convention or using the `folder - files` link. If you start with the `serv` object you use the naming convention to identify the `serv.Doc` object. As in our `JavaDoc` model a `HaveSubUnit` increment covers a `DocFile` object and as the `HaveRoot` increment and command address the same kind of increment, the `HaveRoot` commands needs to edit (i.e. remove) a potentially attached `DocFile` object, too, cf. Listing 1.5.

Note, the `version` attribute of our `serv.Doc` object is not colored as the corresponding `JavaPackage` and the corresponding `HaveSubUnit` command do not have any version information.

To summarize, you start with core objects and collect attributes, links (and neighbors) that form an increment, i.e. that are edited together. Core objects of the same type should have increments of similar structure. For such increments you introduce a command with parameter attributes that correspond to attribute values and link targets. If there are alternative cases, you may use different command types or choose the desired variant from parameter values. This decision must not rely on context properties. For each increment in some source model, you identify a corresponding increment within the target model. Increments shall not overlap and all (relevant) parts of the models shall be covered. Thereby, you construct commands with simple contexts and non-overlapping increments, i.e. commutative commands.

You may check the commutativity of your commands 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 `id`s, comparison of two models is quite easy in our case.

5 Collaborative Editing and Merge Conflicts

Once you got your commands right, you still have to deal with multiple or concurrent edits of the same increment. Assume, you have already executed command (`HaveLeaf Editor serv 1.0`) and now you want to change the version to 1.1. Thus you run the command (`HaveLeaf Editor serv 1.1`). In your current editor 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 editors, concurrently, and you try to synchronize afterwards. As the two commands are conflicting, i.e. they assign different values to the `vTag` attribute of the `Editor`

object, 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, [10] explicitly mentions this problem and puts it in future work. This paper solves this problem with the help of additional time stamps and with the help of command specific merge strategies. 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 && ! command.overwrites(oldCommand)) {
17             return;
18         }
19         command.run(this);
20         activeCommands.put(id, command);
21     }
22     ...
```

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 13[1]. If you serialize a command later on and send it e.g. to another editor, 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` we ask our new `command` whether it is going to `overwrite` the `oldCommand` (Line 16). Listing 1.15 shows the default implementation of method `overwrites`. Line 4 compares the time stamps and we do not

[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]

overwrite if the current command is older than the `oldCommand`. On equal time stamps (Line 6) it is probably the same command received twice and there is no need to execute it again. However, for the unlikely case that two editors concurrently create different commands for the same `id` and with the same time stamp, we do a string compare of the yaml representation of our commands and we do not execute the current command if the oldCommand is lexically later or equal (Line 7 to 10).

```
1  public class ModelCommand {
2      ...
3      public boolean overwrites(ModelCommand oldCommand) {
4          if (oldCommand.getTime().compareTo(time) > 0) {
5              return false;
6          } else if (oldCommand.getTime().equals(time)) {
7              String oldYaml = Yaml.encode(oldCommand);
8              String newYaml = Yaml.encode(this);
9              if (oldYaml.compareTo(newYaml) >= 0) {
10                 return false;
11             }
12         }
13         return true;
14     }
15     ...
```

Listing 1.15. Command execution

If there is no `oldCommand` or if the new `command` overwrites the `oldCommand` our editor executes the new `command` in Line 19 of Listing 1.14. Finally, the new `command` is added to the hash table of active commands (Line 20).

Thus, if you run (`HaveLeaf Editor serv 1.0`) in some editor at e.g. 13:36 o'clock and you run (`HaveLeaf Editor serv 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 the `Editor` object will be `1.1`. Note, our model synchronisation scheme works also for two editors with different meta models.

The default implementation of method `overwrites` shown in Listing 1.15 resolves editing or merge conflicts with a "last edit wins" strategy. While this works quite frequently, in some cases you may want another strategy for conflict resolution. For example in case of a seat reservation system you might want a "first edit wins" strategy. Or in our version number example you may want a "highest version wins" strategy. Therefore, each command may overwrite the inherited default implementation of method `overwrites` and implement its own strategy. If you do this, ensure that all editors (or models) use the same strategy during model synchronisation.

6 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 (core object and 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 (core object and 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 1.16 and [1]. (The `mapOfParsedObjects` will be discussed in Section 7.)

```
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 1.16. 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 1.17 will be discussed in Section 7. Line 11 of Listing 1.17 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 1.17. getObjectFrame method

Similarly, when we want to create a model object explicitly (green rule parts) we call method `getOrCreate` on the corresponding editor, cf. Listing 1.18 . Again, Lines 6 to 10 will be discussed in Section 7. Line 11 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 1.18. getOrCreate method

In order to remove an object from our model we call method `removeModelObject` on the corresponding editor. Line 6 of Listing 1.19 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 1.19. removeModelObject method

Actually, to remove some object from our model, we need to (implement and) call the `remove` 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 1.20.

```
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.remove(editor);
10        }
11        return null;
12    }
```

Listing 1.20. 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 default "last edit wins" strategy 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 1.20 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 `remove` 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 1.20 retrieves the command to be removed from our `activeCommands` and Line 9 calls its `remove` method. Afterwards, the `execute` method of our `editor` replaces the `removed` command with the `RemoveCommand` within our `activeCommands`, cf. Line 20 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.

7 Parsing

Sometimes, you may want to edit your model directly, cf. Line 13 to 24 of Listing 1.21. 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 core model object that gets the same `id` as the command. This core model object becomes the nucleus for each increment. Thus, in our approach the identification of model increments starts with these core 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  private Set changedObjects = new LinkedHashSet();
7  @Test
8  public void testManualChangesAndParsing() {
9      JavaPackagesEditor javaPackagesEditor
10             = new JavaPackagesEditor();
11     startSituation(javaPackagesEditor);
12     registerModelObjectListener(javaPackagesEditor, this);
13     JavaPackage com = new JavaPackage().setId("com");
14     JavaPackage org = (JavaPackage)
15             javaPackagesEditor.getModelObject("org");
16     com.withSubPackages(org);
17     JavaPackage fulib = (JavaPackage)
18             javaPackagesEditor.getModelObject("fulib");
19     fulib.setPPack(null);
20     JavaClass command = new JavaClass()
21             .setPack(fulib).setVTag("1.1").setId("Command");
22     JavaClass editorClass = (JavaClass)
23             javaPackagesEditor.getModelObject("Editor");
24     editorClass.setVTag("1.1");
25     Set allObjects = Yaml.findAllObjects(com, fulib);
26     javaPackagesEditor.parse(changedObjects);
27     ...
28 }
29 ...
30 }
```

Listing 1.21. Command execution

In [10] the parsing of a *HaveLeaf* rule requires that the `JavaPackage` that is reached via a `pack` link has already been parsed either by a `HaveRoot` or by an `HaveSubUnit` rule. Similarly, the parsing of a `HaveSubUnit` rule requires that the `JavaPackage` attached to the corresponding sub `JavaPackage` via a `pPack - subPackages` link has already been parsed. Thus, in [10] 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` leafs. 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 12 of Listing 1.21 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 13 to 24. This allows us to call the `parse` method of our `editor` with just the set of `changedObjects` in Line 26. Alternatively, Line 25 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 1.22 registers all objects that shall be parsed into our `mapOfParsedObjects` (cf. Listing 1.16, 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 1.22). 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 1.22 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 `oldCommand`s that will be overwritten by the execution of a new command, cf. Line 14 to 16 of Listing 1.22. 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 found 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 1.22. Command execution

There is still one little design issue to be discussed. In our example, Line 13 of Listing 1.21 creates the `JavaPackage com`, directly. Our change listener will collect the new `com` object as soon as it is linked to the old model object `org` (Line 16). Thus, parsing will create a (`HaveRoot com`) command. When we execute this (`HaveRoot com`) 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 `com` object. Thus we want `getOrCreate` to retrieve this directly created `com` object and `getOrCreate` should not create a new model object. To achieve this, Line 6 of Listing 1.22 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 1.18). Method `getObjectFrame` work similarly (Listing 1.17).

8 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 of 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 [8] 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 `id`s 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 `id`s. We did so e.g. when using Commutative Event Sourcing for the solution of a model synchronisation case in the Transformation Tool Contest 2020 [6] and [1]. If you use `id`s for cross model referencing, you want to assign these `id`s directly and you do not want e.g. a database system generating your `id`s. Directly edited `id`s 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 `id`s 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 the 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 would not help).

However, we have used Commutative Event Sourcing with great success in [6] 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 [3]. 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. The qvt standard. https://www.omg.org/spec/QVT/, last viewed 31.08.2020
3. Youtube playlist labor microservices, winter term 2019 / 2020. https://www.youtube.com/playlist?list=PLohPa1TMsVqrI0FaMTySbGr02JkQs5MhQ, last viewed 31.08.2020

4. Anjorin, A., Diskin, Z., Jouault, F., Ko, H.S., Leblebici, E., Westfechtel, B.: Benchmarx reloaded: A practical benchmark framework for bidirectional transformations. In: BX@ ETAPS. pp. 15–30 (2017)
5. Copei, S., Sälzer, M., Zündorf, A.: Multidirectional transformations for microservices. In: Proc. Dagstuhl Seminar. vol. 18491 (2019)
6. Copei, S., Zündorf, A.: The fulib solution to the ttc 2020 migration case
7. 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)
8. 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)
9. Evans, E.: Domain-driven design: tackling complexity in the heart of software. Addison-Wesley Professional (2004)
10. 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)
11. 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)
12. 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)
13. Macedo, N., Cunha, A.: Implementing qvt-r bidirectional model transformations using alloy. In: International Conference on Fundamental Approaches to Software Engineering. pp. 297–311. Springer (2013)
14. 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)
15. 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)
16. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Pearson Education (2008)
17. Stevens, P.: Bidirectional model transformations in qvt: semantic issues and open questions. Software & Systems Modeling **9**(1), 7 (2010)
18. Vernon, V.: Implementing domain-driven design. Addison-Wesley (2013)