EXAM NUMBER

Y3864454

TABLE OF CONTENTS

1. METAMODEL

Metamodel for the managing issues is designed for automating the process of communication among team members and it assists to uniformly distribute load as well as to control work process. What is more, it helps to understand the work flow deeply and to solve appearing problems quickly. The basic objective of a metamodel is to provide domain-experts (i.e. three typical DSL stakeholders [1]) with an editor that facilitates creation of software management models. Finally, metamodeling promotes to "support the further development process" [2, p. 101].

According to Paige, "metamodelling follows a well-defined process" [3, p. 402], which is consists of 6 consistent steps. Let's follow these clear instructions. Firstly, we are going to define an abstract syntax. The graphical syntax that conforms to the abstract syntax is displayed in Fig. 1.

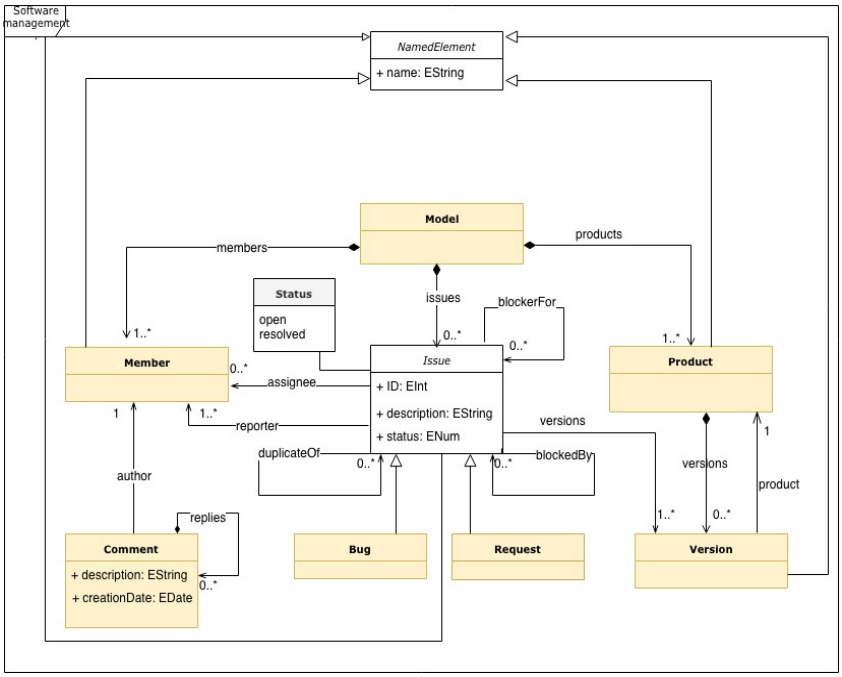

Fig. 1. Metamodel, in Ecore's EMF notation, defining the abstract syntax for Software management

In projectable metamodel, the main entity is a "Model" that consists of several fields – classes feature EReference (by value): Issues, Members and Products. The relationship between class "Model" and each of the three internal entities (Issues, Members and Products) is a one-to-many. However, model must contain at least one Member and one Product and may contain no Issues which is displayed in Fig. 1 as "1..*" and "0..*" respectively.

Since each of four entities (Member, Product, Version, Issue) has name, we define an abstract class "NamedElement" in order to avoid code duplication. Thus, we have an inheritance hierarchie. Another abstract class is "Issue", as we have two types of issues (bug reports and enhancement requests). Classes "Bug" and "Request" are inheritable classes from "Issue". Issues may have no blockers and may not be blocked by other issues, also they may not duplicate other issues. It is shown in Fig. 1 as "0..*"in the end of the rows.

Class "Status" is built on the Ecore type Enumeration and lists two states – open/resolved. Thus, we can mark issue as open or closed. It should be noted, that each comment must have an author which is displayed at the diagram as "1" at the end of the link. In the model we have two cases of opposite references. In the first case, as each product can have several versions, we have class "Product" that has a containment reference to class "Version". Since we want to know a parent product of a specific version, we need to navigate in both directions and, for this reason, we use opposite references. In the second case, as we need to know for each issue its blockers and also blocked issues by this one, we use opposite references in class "Issue". The definition of a graphical syntax for the metamodel is presented in Listing 1.

```
@namespace(uri="softwaremgmt",prefix="")
package softwaremgmt;
class Model{
   val Issue[*] issues;
   val Member[+] members;
   val Product[+] products;
}
abstract class NamedElement {
   attr String[1] name;
}
class Version extends NamedElement{
   ref Product[1]#versions parentProduct;
}
class Product extends NamedElement{
   val Version[*]#product versions;
}
abstract class Issue extends NamedElement{
   attr int ID;
   attr String description;
   attr Status status;
   ref Member[+] reporter;
   ref Member[*] assignee;
   val Comment[*] comments;
   ref Version[+] versions;
   ref Issue[*] duplicateOf;
   ref Issue [*]#blockerFor blockedBy;
   ref Issue [*]#blockedBy blockerFor;
}
enum Status{
   open;
   resolved;
}
class Bug extends Issue{}
class Request extends Issue{}
class Member extends NamedElement{}
class Comment {
   attr String description;
   attr Date creationDate;
   ref Member[1] author;
   val Comment[*] replies;
}
```

Listing 1: The definition of a graphical syntax for the metamodel DSL.

Another alternative design decision that have been considered is to include a non-containment reference to class “Issue” inside class “Product”. As a result, we can find all of the issues of the specific product much more faster. However, it would be impossible to relate one issue with different versions of ther products. Moreover, we could create an abstract class “Person” and make class “Member” inheritable from it, but it is not very important in this domain.

What is more, in order to implement a constraint “bugs can block enhancment requests, but not the other way around” we could make a non-containment reference in classes “Bug” and “Request” like shown in the Listing 2. Thus, bugs cannot be blocked by requests and cannot be duplicates of requests. However, if we followed this structure, it would further contributed to code duplication while creating constrainment through EVL. In other words, we would have to write constraints both for class “Bug” and class “Request”. For this reason, we used opposite references “blockerFor” and “blockedBy” in class “Issue” in order to navigate in both directions. Overall, the metamodel for a domain-specific language and supporting tooling for capturing and managing issues related to software products was designed and successfully implemented in EMF.

```
class Bug extends Issue{
   ref Bug[*] duplicateOfBug;
   ref Bug[*] blockedByBug;
}
class Request extends Issue{
   ref Request[*] duplicateOfRequest;
   ref Issue[*] blockedByIssue;}
```

Listing 2: A non-containment references in classes “Bug” and “Request”.

2. CONCRETE SYNTAX AND EDITOR

2.1 Syntax design and editor implementation decisions

Let's define a concrete syntax which is based on abstract syntax through GMF. A graphical concrete syntax for software products DSL, which is implemented using Eclipse's GMF Eugenia [4, p.14], is displayed in Fig. 2.

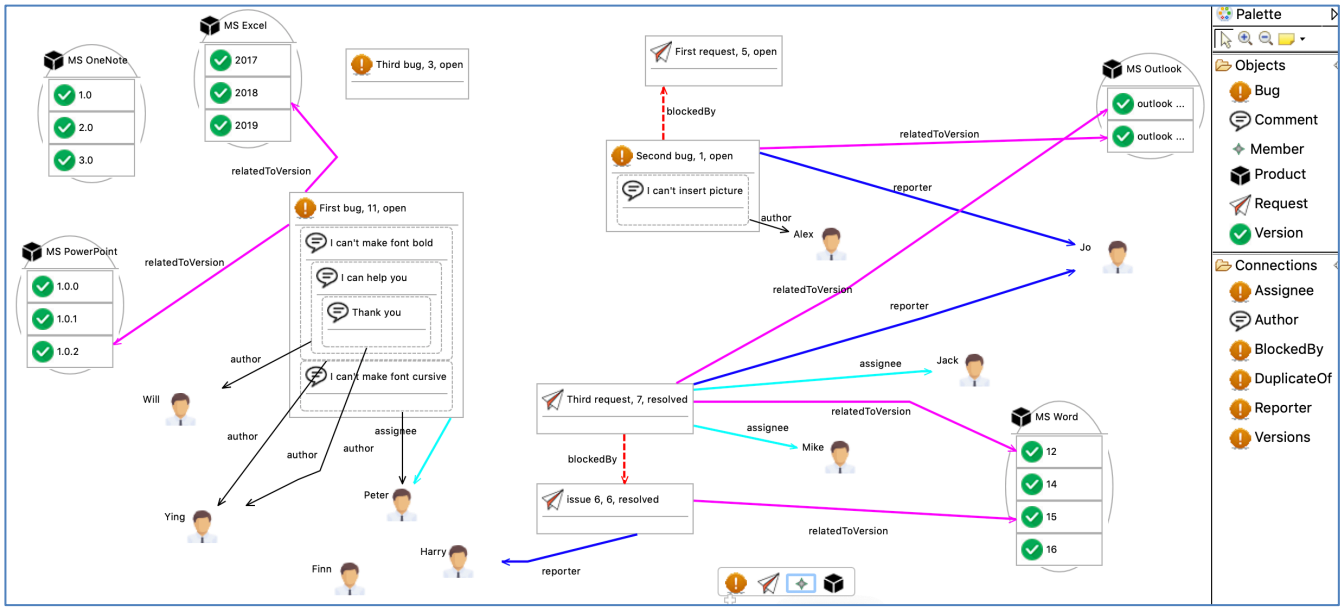

Fig. 2. A graphical concrete syntax implemented using GMF

As we can see from the Fig.2., the graphical syntax conforms to the abstract syntax. A graphical palette includes six objects (nodes) and six connections (links) each of which are connected to a specific object. As Paige emphasizes, "all graphical concepts and relationships are instances of abstract syntax concepts" [3].

In order to make editing process more convenient and apparent for domain experts, each entity has its own different shape. Product has an ellipse shape; Bugs and Requests have a rectangle shape; Comment has a dashed rounded shape and Member has a specific shape – a customized image defined through svg. What is more, each entity has their own icon that allows domain experts to see entities in a coherent and easy to read way.

In order to make possible to place Comments into Issues and Versions into Products, we use *gmf.compartment* [4, p.17] notation which allows us to create a compartment in a concrete object. Thus, domain experts can add comments to issues, add replies to existing comments and add versions to products. Moreover, using attribute "layout" set on mode "list", comments as well as versions, which are displayed as a list, are placed in their compartments coherently and consistently.

In this DSL is important to know the identifier of each issue (they must be unique) and its status (open or closed). For this reason it would be better to display not only an issue title, but also an identifier and status respectively. As a result, domain experts would easily obtain important information about each issue.

To specify relationships between objects, we use special connections that represent classes feature EReference (val, ref). Each of the connections has their own color and type (solid, dashed). Dashed connections implicate problems with issue, i.e. an issue is a duplicate of other issue or has open blocker(s). All these design decisions enable domain experts to distinguish connections easily and create models in a straightforward way.

2.2 Strengths and weaknesses of GMF

Now let's look at the strengths and weaknesses of GMF. We are going to compare GMF with Eclipse Xtext language. Firstly, let's understand the main principles of Xtex.

According to Saad [5, p. 205], Xtex is a special toolbox that allows domain experts to create a set of rules for text-based DSL. The fundamental objective of Xtex is "to allow users without the technical knowledge required to use a general purpose programming language to participate in the development process" [5, p. 205].

1.2.1 Strengths

GMF, in comapison with Xtext, has a number of advantages. Firstly, it visualises model entities in a graphical way which are easy to follow without specific knowledge. According to Baetens, "in GMF a user can only choose between some basic shapes" [6, p.20]. In contrast, Xtex create a symbol editor that is similar to Java language (or any other object-oriented language). As a result, the enduser, who have unclear knowledge of OOL, would have difficulties in a model creating. Secondly, the minimum knowledge threshold for building a concrete syntax with GMF is much less than for building a concrete syntax with Xtex. As Baetens emphasises, "a plus for the enduser (model designer, not the meta model designer), as he can just switch perspectives from modeling to actual coding" [6, p. 20]. On the contrary, in order to create a concrete syntax through Xtext, a developer must primarily understand the principles of EBNF rules, after that he must comprehend Xtext grammar and only after these stages he has sufficient knowledge to start a development of a concrete syntax. Finally, if a diagram element has a problem, a GMF editor shows it as a symbol in the diagram and the error panel specifies what caused the error. Thus, a model designer can "select an entry in the problem view and directly navigate to its counterpart in the graphical editor" [5, p. 343]. In summary, a graphical representation of a metamodel and easy to understand Java-based annotations in editor make the process of model-development not so complicated for model designers.

1.2.2 Weaknesses

On the other hand, GMF approach has a list of apparent weaknesses. First of all, if developers want to make some changes in a meta-model, they are forced to regenerate code and run an application again because an editor for models is not located in the same environment. In other words, it affects the speed of the development process and, as a result, GMF development is more time-consuming in comparison with other editors. The second significant disadvantage of GMF in comparison with Xtex is an impossibility to use of version management systems as "no such feature is included in the standard release" [6, p.21]. As a result, cooperative development of models becomes almost impossible. Another important drawback is that "the GMF-specific models are notoriously hard to get right" [7] because of the huge number of metamodels they conform to. Moreover, in order to implement any constraints, we have to use Epsilon Validation Language (which is good), however, sometimes it is really hard to understand what the problem is. In contrast, Xtext provides us with the opportunity to validate custom constraints and, what is more, it can make automatic quick fixes when error is detected. To summarize, re-running an application after any meta-model changes, an absence of version management systems and difficulties while describing constraints, create a lot of limitations and restrictions while developing through GMF.

1.2.3 Conclusion

In conclusion, in fact "it is hard to tell which one is best because this can be vary due to the users' habits and needs" [6, p.20]. From my own perspective, while choosing which approach to use in definiing a concrete syntax, firstly it would be better to have a clear understanding of the model "scope", its constraints and limitations and also potential users (model and meta model designers) of this syntax.

3. MODEL VALIDATION

Let's implement validation constraints through EVL [7, p. 63] in the context of our model. As each issue must have unique identifier, we define a validation constraint "IdMustBeUnique". Firstly, using a key word "*guard*", which helps us to "limit the applicability of invariants" [7, p. 67], we focus only on issues with defined ID. Secondly, using a key word "*check*" we try to find an issue with the same ID as the ID of the context issue (*self* [7, p. 27]). If we are successful in finding such issue, the size of the sub-collection will be > 0. In that case we print a message that identifiers must be unique. The full code of the constraint is shown in Fig. 3.

```
constraint IdMustBeUnique {
    guard: self.ID.isDefined()
    check: Issue.allInstances().select(i|i.ID=self.ID).size()=1
    message: "Issue `"+self.name+"`  must have unique ID. Now it has ID=" +self.ID
}
```

Fig. 3. The constraint "IdMustBeUnique"

Another validation constraint is to check whether the textual description of an issue is at least 10 characters long "DescriptionMustBeTenLength". The first step is to check whether the description is defined. For this reason, we create another validation constraint called "DescriptionMustBeDefined" that verifies if the description of the issue defined and not *null*. If it fails, the system prints the appropriate message and with the help of keyword "*fix*" allows users to change description to a line "standart description". As the constraint "DescriptionMustBeTenLength" depends on "DescriptionMustBeDefined", we use keyword "*guard*". After the description is defined, the system checks the length of the specific description and if it less than 10 characters long, prints the relevant message. The full code of these constraints is shown in Fig. 4.

```
context M!Issue {
    constraint DescriptionMustBeDefined{
        check: (self.description.isDefined()==true and self.description<>null)
        fix {
            title: "The description of the issue `" +self.name+"` is undefined. Change description to `standart problem`"
            do {
                self.description="standart description";
            }
        }
    }

    constraint DescriptionMustBeTenLength {
        guard: self.satisfies("DescriptionMustBeDefined")
        check: self.description.length()>=10
        message: "The description of the issue `" +self.name+"` must be at least 10 characters long"
    }
```

Fig. 4. The constraints "DescriptionMustBeDefined" and "DescriptionMustBeTenLength

To implement the constraint "an issue cannot be a duplicate of itself", we use an operation "*excludes*" which returns true if the collection excludes the item [7, p. 38]. Another similar constraint is "an issue cannot be blocked by itself" which inherits the same principal as in the constraint described above. To be sure that a closed issue does not have open blockers, we create two constraints "isResolved" and "CanNotHaveOpenBlockers" which depend on each other. The first validation constraint helps us to identify whether issue is open or closed. After we definitely know that an issue is closed, we check the number of its blockers and if it more than 0, the system prints the appropriate message. To address the constraint "Bugs can block enhancement requests, but not the other way round", firstly, we need to check that the blocked issue is "Bug". After that we can check whether the blocker issue is a "Request" (to do it we use an method "*isEmpty()*") and if it's so, the system prints message describing the problem.

Since each issue must have name, we define a constraint "IssueMustHaveName" that helps to achieve that. Firstly, the system checks whether a name of the concrete issue is defined and not *null,* and if it so, it offers users to change it to a line "issue" and number of all issues +1. Thus, each issue will have a name.

And the last significant constraint is to exclude "direct or indirect blocking cycles". To solve that, we define two constraints "hasBlockers" and "NoBlockingCycle" which depend on each other. First of all, we check whether an issue has blockers and if it is true, with operator "*forAll"* we compare blockers of the specific issues with blocked by this issue issues. Thus, we can guarantee that in the model are not

direct or indirect blocking cycles. The full code of the described constraints can be found in the source code.

Let's integrate the implemented constraints with the graphical editor which is shown in Fig. 5. As we can see from Fig. 5 and Fig. 6, the editor errors and warnings appear in the graphical editor as well as in the console. For example, as we can see from the Fig. 5, the Issue "Third Bug" has a red cross in the right top. It means that this Issue has problems: the description should more than 10 characters long and it cannot be blocked by itself. Also we can make some fixes through invariants (change description to a pre-defined line "standard description" if it is not defined). Another example is "issue 6" and "third request". In here we can see blocking cycle which is not allowed in this model. As a result, red crosses appear in the right top of each issue which signifies graphical designers about errors in the model.

In conclusion, developed validation constraints help us to make a model consistent and coherent and also assist us to capture more complex validation rules that the metamodel cannot express.

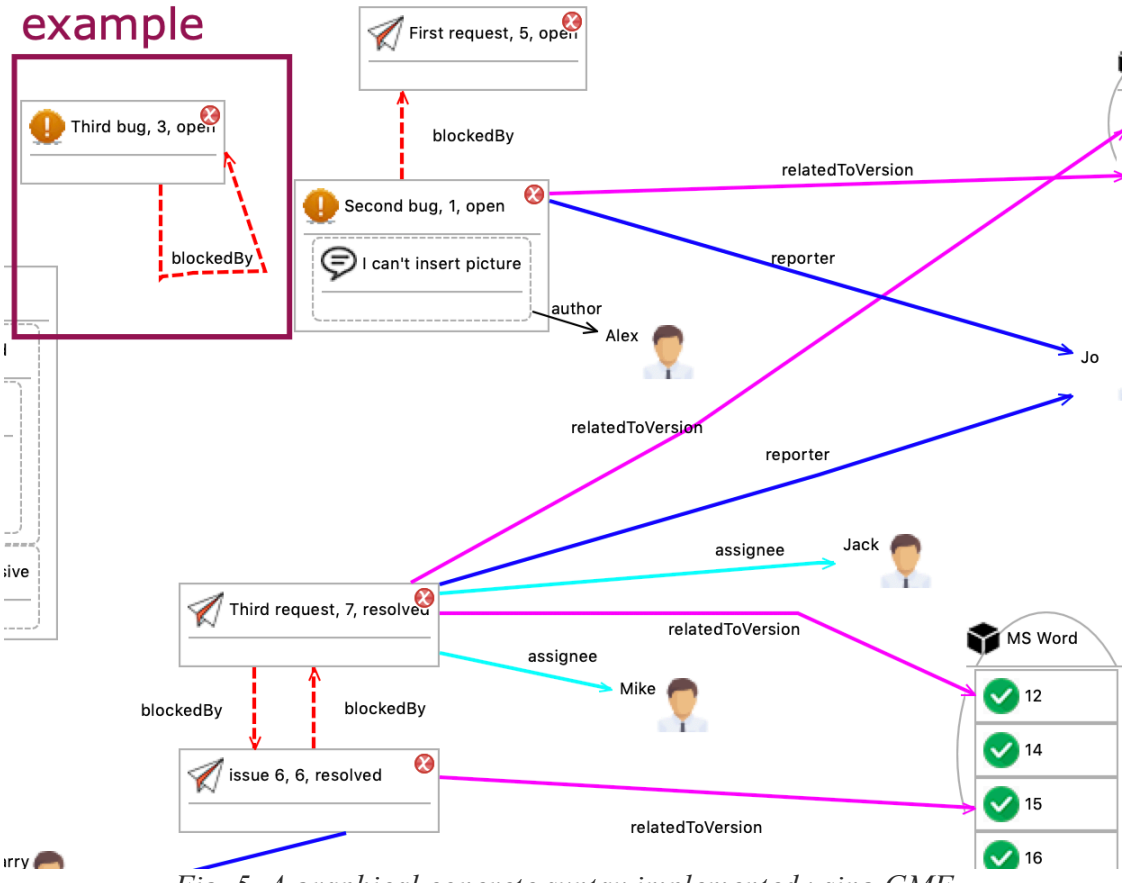

Fig. 5. A graphical concrete syntax implemented using GMF

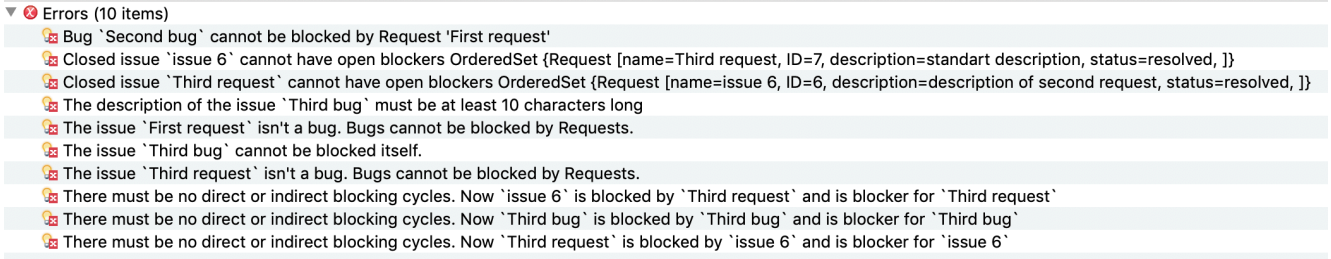

Fig. 6. In-editor errors

4. MODEL-TO-TEXT TRANSFORMATION

With the help of Epsilon Generation Language (EGL) [8], we can transform models into a number of different text-based artefacts [7, p. 105]. Let's generate a set of hyperlinked HTML pages in EGL. For each page we are going to create a specific template that describes the structure of the future HTML page – file_name.egl. For example, the template for the page, which is intended for viewing the details and comments of a specific issue, is shown in Fig. 7. The information is presented in a table because it is easier to read data in such organisation. As we can see from the figure, using a loop `for` we insert each comment in the table and, thus, shape HTML page. Similarly, we create other templates for different HTML pages. Each page starts with a header which describes the main purpose of the page; each page has a table where we can find information on a specific problem.

```
<h1>The details and comments of issue [%=i.ID%]: [%=i.Name%]</h1>
<p>Description: [%=i.Description%]</p>
<p>Status: [%=i.Status%]</p>

<h2>Comments</h2>
<table border="1">
    <tr>
        <td>Description</td>
        <td>Author</td>
        <td>Creation Date</td>
        <td>Parent Comment</td>
    </tr>
    [%for (comment in i.Comments){%]

    <tr>
        <td>[%=comment.Description%]</td>
        <td> [%=comment.Author.Name%]</td>
        <td>[%=comment.creationDate%]</td>
        <td>This comment is root and has no parent.</td>
    </tr>
    [%if (comment.replies.size()>0){%]
        [%for (reply in comment.replies){%]
         <tr>
            <td>[%=reply.Description%]</td>
            <td> [%=reply.Author.Name%]</td>
            <td>[%=reply.creationDate%]</td>
            <td>[%=comment.Description%]</td>
        </tr>
        [%}%]
    [%}%]
    [%}%]

</table>
```

Fig. 7 The structure of the HTML page for viewing comments

After we create all of the templates, we need to coordinate language for EGL templates using EGX. To generate HTML page, we should create a rule in EGX which describes a template's name, a name of a target file and check the presence of constraints with a keyword *"guard"*. With keywords *"pre"* and *"post"* we let a user know when generation starts and when it finishes – appropriate notifications will appear in console.

After generation process completed, we can see generated HTML pages in a folder *"gen/"*. Generated pages for the graphical model, which is presented in Fig. 5, are presented in the left part of Fig. 8.

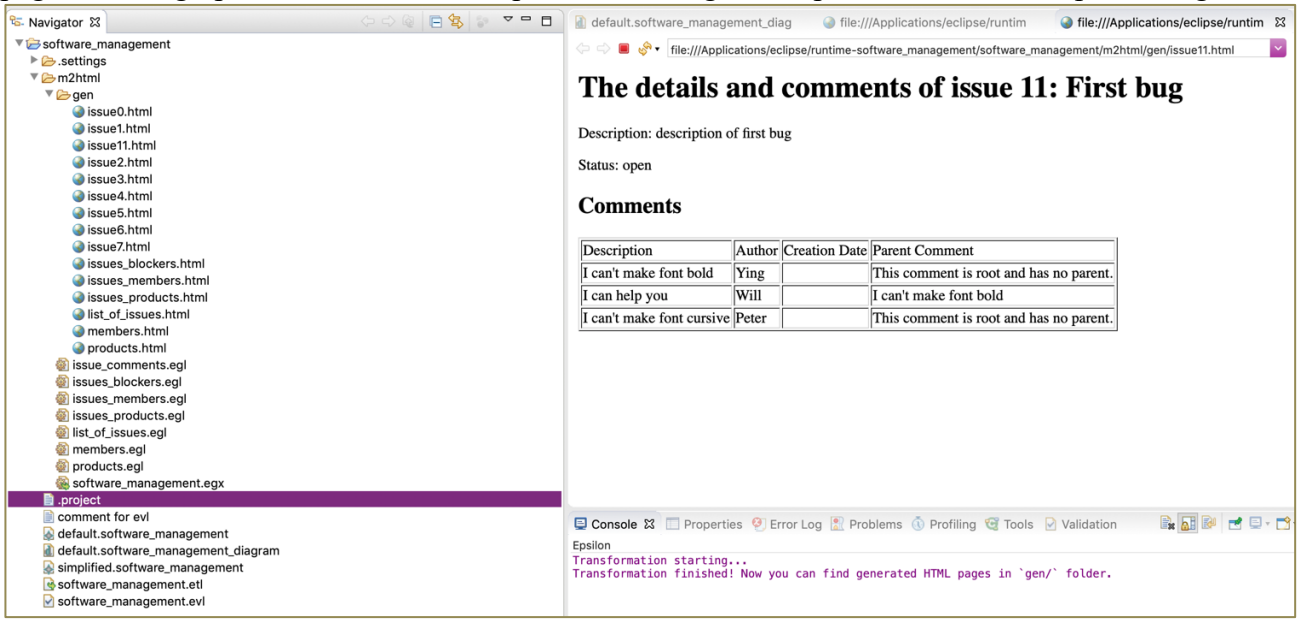

Fig. 8. A generated HTML-page for viewing comments and details of the specific issue

As we can see from Fig. 8, we have a page named "*list_of_issues*" that contains a links to each issue. Each issue has its own page which describes the details and its comments. In the page of a specific issue (as an example, in the right side of Fig. 8 we can see the page for issue with ID 11 and name "First bug"), we have a table with four columns which are "Description", "Author", "Creation date", and, as authors can add comments to issues and replies to existing comments, we have a column called a "Parent Comment". This column makes it possible to know a parent of a specific comment and, thus, we can easily find replies to a concrete comment.

To view versions of products related to a specific issue, we should go to the file "*issues_products*". Since each issue may relate more than one version of different products, we have a collection of version displayed here in square brackets. In the first square brackets we have names of products and in the second square brackets we have names of the product's versions which are related to this concrete issue.

To view assignees and reporters of each issue, we should go to the file "*issues_members*". To obtain information about issue blockers, we should open file "*issue_blockers*" where we can find a list of blocking issues appearing in order of importance. To sort issues in order of importance, we calculate how many issues are blocked by current issue and, as method *sortBy* returns a copy of the collection in ascending order, we invert returned collection and display it in the screen.

From my own point of view, the usage model-to-text transformation approach has a number of clear advantages:

- As we have the same metamodel for all text artifacts such as HTML pages, LaTeX, RTF, we have a really consistent and coherent application. In other words, we don't have different information in some sources and we can be sure that presented information is not ambiguous and, thus, it is consistent and clear in every generated resource.
- While developing an application with different interfaces, we can significantly save time because different text artifacts are generated automatically from one metamodel quickly.
- If we need to change something, we have to do that only in one place – metamodel. As a result, information will be updated in all interfaces simultaneously which contributes to time saving and makes an application development faster and maintainable;
- Developers don't have to write a lot of code because it generates automatically. In other words, in industry we can save budget money because we need to hire less developers.

5. MODEL-TO-MODEL TRANSFORMATION

Model-to-model transformation has a list of apparent advantages:

- when a metamodel evolve, we can migrate one model to others;
- we can transform an input model in different ways into a number of output models of different modelling languages [7, p. 83]: in tables, in lists and others. In other words, we can observe our model from different sides.
- having model-to-model transformation, we can create different model-to-text transformations and, thus, have the same information presented in different ways according to an initial purpose.
- this approach can significantly save development time because people don't have to write code manually - it is created automatically.

Let's transform the model, generated with GMF (shown in Fig. 9), to a simplified model which excludes closed issues as well as team members and products that are only associated to such issues. Our new model should conform to the same metamodel.

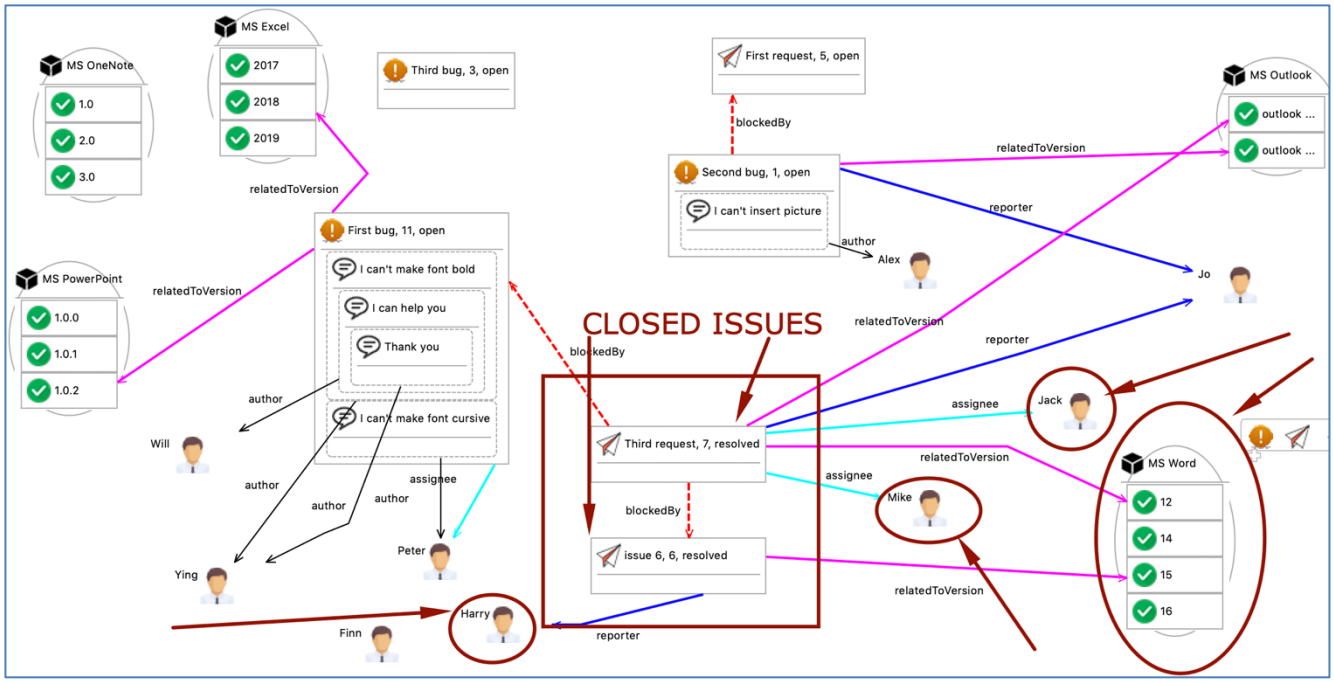

Fig. 9. A model for model-to-model transformation

We are going to use the Epsilon Transformation Language (ETL) [7, p. 83]. According to Sendall, a language for model-to-model transformation should have significant characteristics: "it should be easy-to-understand, yet precise and unambiguous, it should be concise and easy-to-modify, yet complete" [9, p. 8]. From my perspective, Epsilon Transformation Language satisfies all characteristics emphasized by Sendall.

First of all, to let user know that a generation begins, we are going to create special notifications that will appear in console using keywords *"pre"* and *"post"*. As both models conform to the same metamodel, we need to transform only one root entity which is "Model". To achieve that, we create a rule "Model2SimplifiedModel*"* and in here we specify a source model ("InitialModel") and a target model ("SimplifiedModel"). Having done that, we need to exclude closed issues, members and products that are only associated to closed issues. We are going to follow the plan below.

1. Find open and closed issues.
2. Find members which are only associated to open issues and add them to the collection "*members*".
3. Find members which are nor assignees neither reporters of any issue (they don't take part in any issue) and add them to the above defined collection "*members*".
4. Thus, we exclude members which are only associated to closed issues.

5. Find versions of products which are only associated to open issues and add them to the collection "*products*".
6. Find versions of products which don't have open issues and add them to the above defined collection "*products*".
7. Thus, we exclude products which are only associated to closed issues.
8. Finally, we add three collections which are "*open issues*", "*members*", "*products*" to a new simplified model.

In order to implement the constraint "the transformation must not modify its source mode", we edit run configurations as follows: we want to read the initial model on load, but don't want to store on disposal; we want to store the new simplified model on disposal, but don't want to read on load. With such run configuration we can be sure that the transformation will not modify its source code.

Now we can generate our new simplified model for the model shown in Fig. 9. The initial model and the generated model in a treeview format are shown in Fig. 10 and Fig. 12.

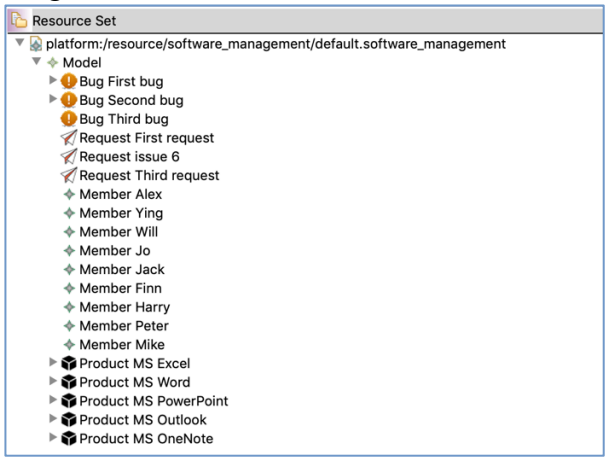

Fig. 10. Initial model

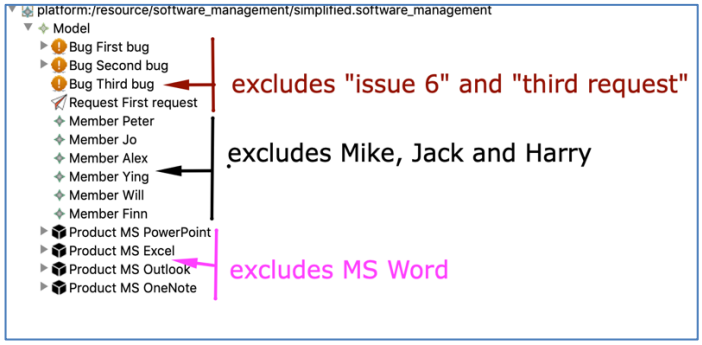

Fig. 11. Generated model

As we can see from Fig. 10 and Fig. 12, a new model exclude resolved issues named "*third request*" and "*issue 6*". Moreover, it also excludes team members Jack, Mike and Harry because they are only associated to issues "*third request*" and "*issue 6*" as well as the product named "*MS Word*".

6. LIST OF RESOURCES

1. Kolovos, D.S., Paige, R.F., Kelly, T. & Polack, F.A. Requirements for domain-specific languages. In Proc. of ECOOP Workshop on Domain-Specific Program Development (DSPD), 2006.
2. Thomas Stahl et al., Model-Driven Software Development, Addison-Wesley, 2006.
3. R.F. Paige et al. Science of Computer Programming 96, 2014.
4. Kolovos, D.S., Rose, L.M., Paige, R.F. & Polack, F.A.C. Raising the level of abstraction in the development of GMF-based graphical model editors. In ICSE Workshop on Modeling in Software Engineering, MiSE 2009, Vancouver, BC, Canada, May 17-18, 2009.
5. Saad C., Bauer B. (2013) Data-Flow Based Model Analysis and Its Applications. In: Moreira A., Schätz B., Gray J., Vallecillo A., Clarke P.
6. Baetens, N.: Comparing graphical DSL editors: AToM3, GMF, MetaEdit+. Technical report, University of Antwerp (2011).
7. Dimitrios S. Kolovos, Louis M. Rose, Richard F. Paige. The Epsilon Book. 2008. https://www.eclipse.org/epsilon/doc/book/
8. Dimitrios S. Kolovos, Louis M. Rose, and James R. Williams Using Model-to-Text Transformation for Dynamic Web-based Model Navigation, Technical report, University of York.
9. Shane Sendall, Wojtek Kozaczynski Model Transformation – the Heart and Soul of Model-Driven Software Development, Technical report, Swiss Federal Institute of Technology in Lausanne (EPFL).