Arab J Sci Eng (2015) 40:1359–1373
DOI 10.1007/s13369-015-1607-z

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

A Technique for Metamodeling Diagram Types with Tool Support

Iván García-Magariño · Guillermo Palacios-Navarro

Received: 16 October 2014 / Accepted: 4 February 2015 / Published online: 15 February 2015
© King Fahd University of Petroleum and Minerals 2015

Abstract Some domain-specific modeling languages (DSMLs) use different diagram types but these are not explicitly included in metamodels. The definition of diagram types is an increasing demand for some computer-aided software engineering tools for DSMLs. The current work presents a technique that allows designers to define diagram types of DSMLs with metamodels in a straightforward and tool-supported way. This technique also facilitates the processing of models when some elements appear in several diagrams, by including a single dictionary of all the entities and their relationships. The presented technique is supported with a novel tool called diagram-type editor tool (DTET). DTET receives input from a DSML metamodel and allows designers to define a set of diagram types with a graphical user interface. Then, DTET generates a metamodel with these diagram-type definitions. For the evaluation, 39 testers from five different countries (Spain, Dominican Republic, Colombia, Ecuador and France) experienced the presented approach and other alternatives, measuring the times of definition and the numbers of mistakes. The results show that the presented technique and DTET are faster and less error-prone for the definition of diagram types than the alternatives with statistically significant differences.

Keywords Model-driven engineering · Metamodel · Diagram type · Domain-specific modeling language · Computer-aided software engineering tool

I. García-Magariño (✉)
Department of Computer Science and Engineering of Systems, University of Zaragoza, Teruel, Spain
e-mail: ivangmg@unizar.es

G. Palacios-Navarro
Department of Electronic Engineering and Communications, University of Zaragoza, Teruel, Spain
e-mail: guillermo.palacios@unizar.es

1 Introduction

Domain-specific modeling languages (DSMLs) are useful for modeling plenty domain-specific areas, and as much specific as, for example, the Hajj pilgrimage traffic scenario [1] or the model predictive control of the speed of a permanent synchronous motor [2]. Each DSML has its own specific modeling elements. Some DSMLs represent diagrams that share some modeling elements. Some DSMLs also use different types of diagrams. In this manner, the diagram types can have restrictions on the elements that are allowed to be included and, consequently, can guide users in designing domain-specific models. For example, there can be diagram types concerning requirements, structural design, behavioral design, implementation and testing. Each of these categories of diagram types can cover a great amount of possibilities. For instance, there are several diagram types regarding requirements, as for example the ones necessary for solving the hierarchy of problems proposed by Memon et al. [3] for requirements engineering education.

The object management group (OMG) defines the UML-diagram interchange (UML-DI) [4] specification, which provides a mechanism for interchanging diagrams of the unified modeling language (UML). However, UML-DI cannot be applied to DSMLs because UML-DI is intended for UML, and most DSMLs use modeling elements that are different from the UML elements. In addition, UML-DI uses the cutouts when some diagrams share some elements. The processing of these cutouts is arguably complex, and there are undetermined situations when, for example, removing elements of a cutout that is viewed from another cutout. These cutouts also make it difficult other operations such as the recognition of design patterns with their variations like in the proposal of Rasool and Mäder [5].

The present approach considers the connection-based modeling languages, to which most current modeling languages belong, instead of only supporting UML. A general entity and a general relationship are considered for each DSML and can be extended with its specific entities and relationships. The current approach defines diagrams as the views of models. The presented technique indicates the elements that are necessary for defining diagram types, which can only contain certain element types. The diagram types are defined in a metamodel different from the one for the DSML. Moreover, the technique proposes that the designer defines the DSML diagram types through the graphical user interface (GUI) of DTET. In this way, the designer indicates the names of the diagram types and associates these with certain element types. Then, DTET automatically generates a metamodel with the diagram types.

The proposed technique and DTET can be applied to most DSMLs because new metamodels are created for defining diagram types without altering the metamodels of DSMLs. The processing of models is facilitated with the present approach, because all the entities and relationships of a whole model are included in a global dictionary without duplicates and interferences with diagram-specific issues. In this manner, entities and relationships can be processed regardless of their organization into diagrams.

The experimentation of the current work shows that DTET is faster and more reliable for defining diagram types than the analyzed alternatives. These experiments are based on the experience of 36 postgraduate testers and three PhD testers and concrete measurements such as the duration times and numbers of mistakes. In particular, this work selects the Eclipse modeling framework (EMF) editor and a text editor as alternatives for the experimentation, because these (1) use the same metamodeling language as the current approach and (2) are popular.

The remaining of the paper is structured as follows. The next section describes the motivation of the current approach. Section 3 introduces metaobject facility (MOF) and EMF, as basis for understanding the present approach. Section 4 presents the technique for defining diagram types with metamodels, and Sect. 5 describes its tool support, i.e., DTET. Section 6 evaluates the current approach with a quantitative comparison with the alternatives. Section 7 discusses the results of this work, comparing these with other related approaches and tools from a qualitative point of view. Finally, Sect. 8 depicts the conclusions and future work.

2 Motivation

The way of defining diagram types can be relevant for computer-aided software engineering (CASE) tools related to modeling languages, because (1) these definitions influence the way diagrams are interchanged among CASE tools, (2) the structure of these diagram-type definitions influence most of the processing in the tools such as the addition/removal of elements, refactoring of some parts, copy operations, design patterns recognition [5] and measuring models (e.g., for predicting their stability [6]), and (3) practitioners can understand how diagrams are managed to foresee its implications. As a proof of all these necessities, the international OMG organization (in charge of several international standards) provides a standard that defines diagram types for UML (one of the most widespread modeling languages), and this standard is called UML-DI.

Nonetheless, there is a gap in the literature regarding the definition of diagram types of DSMLs that are non-UML languages. Thus, each CASE tool of these non-UML languages is usually implemented with its own ad hoc way of defining diagram types. These ways of defining diagram types are usually non-public to the best of authors' knowledge, and consequently, new developers do not know how to face the definition of diagram types when defining a new CASE tool for a DSML. In addition, some CASE tools of the same modeling language cannot interchange diagrams because of the absence of formal definitions of diagram types among other obstacles.

In particular, the current approach proposes a technique and a tool for defining diagram types of connection-based modeling languages (languages with entities connected through relations), which is one of the main kinds of modeling languages. The main kinds of modeling languages are described in the classification of Costagliola et al. [7]. Thus, the current work provides a solution for defining diagram types for a group of languages that is not covered by the existing solution (i.e., UML-DI). This group of languages is the non-UML connection-based DSMLs. It is worth being aware that this group is numerous and it is increasing steeply, because of among other things the recommendations of Model-Driven Engineering (MDE) that encourage to define new DSMLs.

This group contains DSMLs for domains such as chemical compositions, multi-agent systems (MASs), databases, software & system processes, electronic circuits and digital electronic diagrams. Each of these domains has one or several languages supported by one or several tools, in which the current approach can be applied. A few examples of these tools are MarvinSketch, ChemSketch and ChemDraw (for chemical compositions), Prometheus Design Tool, Ingenias Development Kit (IDK), Adelfe and Gaia (for MASs), ER Diagram Tool and Database Design Toolset (for databases), Visio, Eclipse SPEM Designer, and APES2, (for software & systems processes), CircuitLab, NgSpice, LTSpice, MultiSim and TINA (for electronic circuits), and CircuitLogix, TopSpice and Micro-Cap 10 (for digital electronic diagrams). These tools could apply the current approach for defining dia-

gram types in their next versions, and similar future tools will be able to apply the current approach.

Moreover, the current approach has some advantages over UML-DI for UML tools. The first advantage concerns the facilitation of some processing tasks in the entities and relations of a model, avoiding redundant operations when some entities and relations appear in several diagrams. The second advantage is the avoidance of ambiguous situations of some processing because of the cutouts of UML-DI. These advantages are further discussed in Sect. 7.1, and consequently, this discussion is omitted here for the sake of brevity.

As an example of a tool that needs defining diagram types, Fig. 1 shows a snapshot of the IDK tool [8]. This tool uses a language specifically designed for modeling MASs. In the pop-up menu, one can observe that users can add 10 different diagram types, and packages for organizing diagrams. In particular, this figure shows a diagram of the Interaction Model type. The center palette shows all the kinds of modeling entities that are allowed to be added in these diagram types with the corresponding buttons. Specifically, the Interaction Model diagram type can contain 16 different kinds of entities (e.g., Roles, Interaction Units and Tasks). The current approach is useful for efficiently metamodeling all these diagram types, each of which with several kinds of entities and relationships, in a methodological and tool-supported way. In a similar manner, this approach is also useful for overcoming the same problem in tools for other emergent DSMLs with several diagram types.

3 Introduction to MOF and EMF

This section introduces MOF specification and EMF as background of the current work. It also mentions the reasons why EMF and ECore language were selected for defining diagram types with the current approach.

MOF specification [9] defines a metamodel as the definition of a modeling language. MOF uses a metadata architecture consisting in four layers. These four layers are M3, M2, M1 and M0. To begin with, M3 contains meta-metamodels, which define metamodeling languages. For example, MOF and ECore languages are defined with meta-metamodels. M2 contains metamodels, which define modeling languages. This layer is the main scope of this paper. M1 has models, which are instances of metamodels. Finally, M0 contains data, which are instances of models.

EMF [10,11] is a metamodeling framework distributed with the Eclipse development environment. It uses the *ECore* metamodeling language. As a brief introduction to the ECore language, a list of the most relevant ECore elements follows:

– *EClass*. It contains EAttributes and EReferences. The instances of EClass can extend other instances of EClass. In this paper, EClasses are denoted as classes.
– *EAttribute*. It has values of primitive types, such as integers, strings and characters. In this paper, EAttributes are referred as attributes.

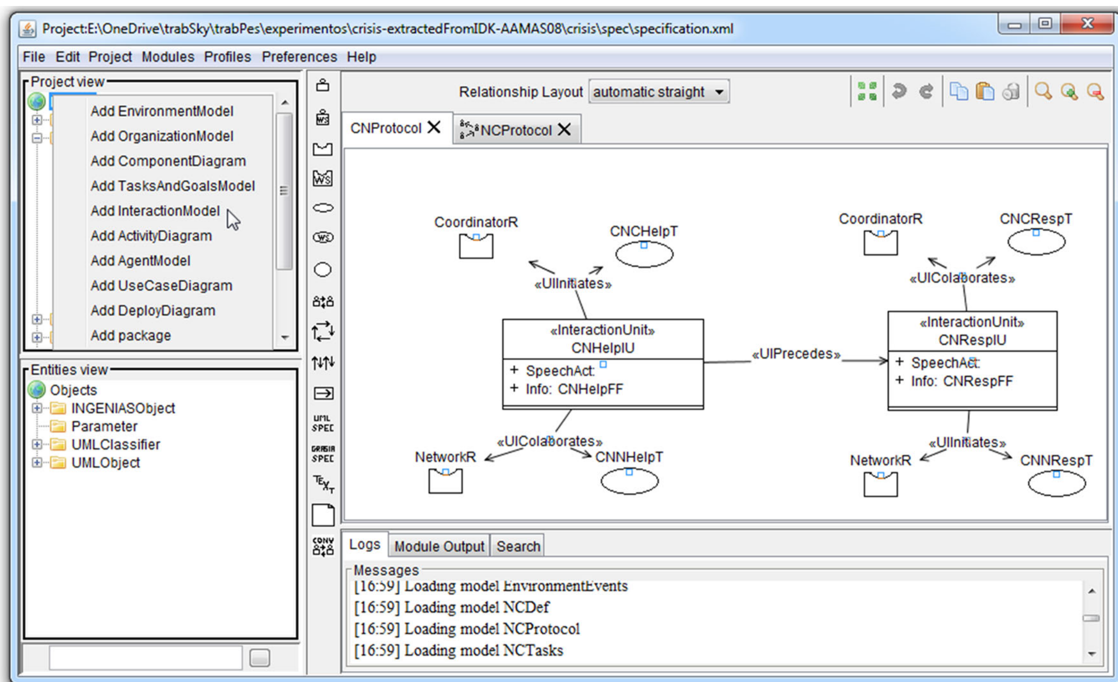

Fig. 1 Snapshot of IDK tool as a motivating example for defining diagram types

- *EReference*. It represents a binary relationship between two EClasses. The instance of the first EClass contains an instance of the EReference. The second EClass is the EType of the instance of the EReference. Each EReference can be either *multiple* or not. In the former case, the EReference can be instantiated several times in the M1 layer. The EReferences are denoted as references. There are two kinds of EReferences:
  - *containment*: In the model, the XML element representing the container contains a new XML element that represents the referenced object.
  - *non-containment*: In the model, the XML element representing the container just contains a path to the XML element representing the referenced object. The referenced object can be either in another position of the document or in another document.
- *EPackage*. It represents a package of elements of the metamodel. In this article, EPackages are referred as packages.

EMF provides libraries for creating metamodels from programming code, in particular from the Java programming language. These libraries are very useful for creating applications that process or create metamodels. This feature is one of the main reasons for selecting the ECore language for the metamodels used by DTET.

In addition, EMF can automatically generate an editor for a modeling language from its metamodel. For instance, DTET create a metamodel with a list of diagram-type definitions. Then, EMF can automatically generate an editor from the metamodel created by DTET. In this manner, the users can easily have an editor of a modeling language with several diagram types defined by the user themselves. In conclusion, EMF can generate editors from metamodels. This fact is another main reason for selecting EMF and ECore language for the present approach.

4 Technique for Metamodeling Diagram Types

This section presents a technique for metamodeling diagram types. Section 4.1 presents the architecture for defining diagram types. This architecture relies on the specific tool support called DTET. Section 4.2 mentions the modeling language requirements for being an input of DTET. Section 4.3 introduces a guide for adapting metamodels of modeling languages to be supported by DTET. Section 4.4 indicates the basis of the diagram definitions. This basis considers the diagrams as views of a shared dictionary. Finally, Sect. 4.5 describes the metamodels generated by DTET.

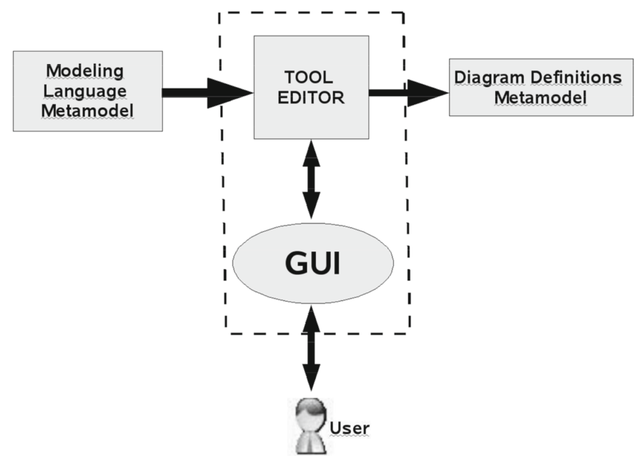

Fig. 2 Architecture of the presented technique

4.1 Architecture of the Technique

Figure 2 shows the architecture of the presented technique for defining diagram types of modeling languages. The present architecture is based on DTET. This tool receives input from a metamodel of a modeling language, which is usually a DSML. This metamodel is selected by the user from the File menu. From now on, this metamodel is referred as *modeling language metamodel*. This metamodel must use the structure indicated in Sect. 4.2. Any metamodel can be adapted to this structure by means of the preadaptation phase described in Sect. 4.3. DTET loads all the entities and relationships from the modeling language. This loading of entities and relationships is necessary for showing these to the user afterward.

DTET provides a GUI, in which the user can create several diagram types. In this section, the *diagram types* are referred as *diagrams* for simplification. The entities and relationships of the modeling language are presented to the user. Both entities and relationships are presented with the entities and relationships hierarchies of the modeling language. The user must link each diagram with entities and relationships of the modeling language. After this, the user can save the definitions of the diagrams, which are saved into a metamodel. From now on, this metamodel is referred as *diagram definitions metamodel*.

In addition, DTET can also load some diagram definitions. In particular, DTET loads the diagram definitions metamodels that were previously created by itself. This feature makes it easier the maintenance of the diagram definitions. This maintenance is especially useful for CASE tools that use a non-stable subset of a modeling language.

Both the modeling language metamodel and the diagram definitions metamodel together (see Fig. 3) represent one metamodel. This metamodel contains the two corresponding internal metamodels. This metamodel can be used for generating a CASE tool. This CASE tool considers both the modeling language and the diagram definitions.

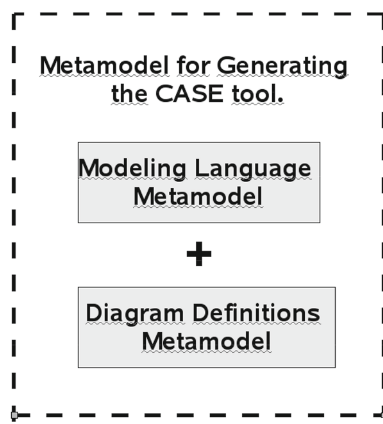

Fig. 3 The metamodel for generating the CASE tool

Keeping the two aforementioned internal metamodels separately has the following advantages:

- *The modularity increases*. The same modeling language metamodel can be combined with several diagram definitions metamodels.
- *The model processing is facilitated*. Since there are two internal metamodels, the models in M1 layer can also be defined with two internal models. Each model is an instance of each metamodel. These models are called the *dictionary* and the *diagrams model*, respectively. The dictionary can be processed automatically by a tool ignoring the diagrams. The diagrams are considered views of the model in this work (read further in Sect. 4.4).
- *A new subset of the modeling language can be defined by means of a new diagram definitions metamodel*. In this case, not all the modeling language elements are included in the diagrams. Sometimes, some CASE tools only implement a subset of a modeling language.

4.2 Modeling-Language Requirements

DTET can define diagrams types mainly for *connection-based* [7] languages. As our previous work [12] shows, most current modeling languages belong to this kind. Nevertheless, the metamodel structure of the modeling languages sometimes need to be adapted for being automatically interpreted by DTET. As discussed in our previous work [12], all connection-based modeling languages can be represented with the required structure (see Fig. 4) by means of the guidelines mentioned in that work. Basically, the modeling language elements are classified into *entities* and *relationships* and placed in the following structure:

- A package named *entities*. This package must contain a class for each entity. All the classes in this package must represent entities of the modeling language. The entities must be organized into a hierarchy. The classes extend other classes in a tree hierarchy.
- A package named *relations*. The classes within this package must represent all the relationships organized into a hierarchy. The classes within this package can only represent relationships.
- A package named *specification*. It must contain a class called *Specification* (see Fig. 5). The instance of this class must be the root element of the M1 model. This class must have two *multiple containment* references. One reference must point to the root of the entity hierarchy, and the other must point to the root of the relationships hierarchy. In this manner, the instance of the Specification class can contain any entity or relationship at M1 level.
- Other packages. These packages can contain classes of any type. For instance, the example of the Fig. 4 has an additional package, which is called *association_ends*. This package contains a class for each relationship end. The goal is to define attributes in the relationship ends. These relationship ends are called *roles* in the metamodel literature [13].

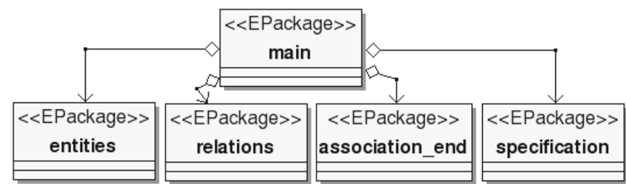

Fig. 4 The modeling-language metamodel structure

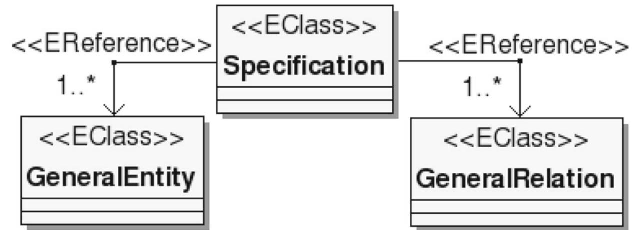

Fig. 5 The *Specification* class

It is worth mentioning that, if the spatial positions of diagram elements are represented in another metamodel, this technique and DTET can also support *Geometric-based* modeling languages [7] (in this case, all the elements are classified as entities) and hybrid modeling languages between connection-based and geometric-based.

4.3 Pre-adaptation Phase

For being possible that DTET supports a metamodel of a CASE tool, it is usually necessary that practitioners have to perform a preadaptation phase. In this phase, practitioners must follow the following steps:

1. *Translation to ECore*: This step is only necessary in metamodels that are not already defined in ECore. If necessary, practitioner must translate the metamodel from the corresponding metamodeling language to ECore. This translation is encouraged to be performed automatically to save effort and time. Our previous work already shows how to perform translations between different metamodeling languages [14], and it is omitted here for the sake of brevity.
2. *Classification of entities and relations*: Based on the fact that the current approach is applied to connection-based languages (as stated in Sect. 4.2), by definition the language should have entities and relations. This step classifies the existing metamodeling elements between the categories (1) entities, (2) relations and (3) others.
3. *Adaptation of entities*: Practitioners observe whether all the entities are within the same package. In this case, the package must be safely renamed as *entities* (by refactoring with the EMF Eclipse tool). Otherwise, a new package called entities must be defined, and all the entities must be safely moved to this package. Usually, all the entities are within a hierarchy with a single root. If it is not the case, practitioners define a new general entity, and all the entities without parent must be established as children of this new general entity.
4. *Adaption of relations*: Similarly to the previous step, if all the relations belong to the same package, this must be safely renamed. Otherwise, all the relations must be safely moved to a new package called relations. If there is not a hierarchy of relations with a single root, all the relations without parents must be set as children of a new general relation.
5. *Definition of the specification*: Find the element that is the root in M1 level, and observe whether it has multiple references to the roots of the entity hierarchy and the relation hierarchy. If not, practitioners must define these references. Then, this element must be safely renamed as Specification and placed in a package called specification.

Once the aforementioned steps have been followed, practitioners can load the corresponding metamodel of the modeling language from the File menu of DTET. With this preadaptation phase, DTET can assist practitioners in defining the diagram definitions metamodels for different CASE tools.

4.4 Diagrams Modeled as Views

Models of systems sometimes are huge. Huge models are usually difficult to be understood by designers. Hence, division of models in smaller pieces is quite common in the CASE tools history (Rational Rose, BoUML, ArgoUML and Together).

In the current approach, the model of a system is considered to be an indivisible whole since all their elements are usually connected between each other. Then, the diagrams are considered *views* of the whole but not parts of it.

The *view* term is used in the database terminology, where the user can see only some rows of a database table. The view term is also used in the software design pattern terminology. For instance, the *Model-View-Controller* design pattern determines that a view is a piece of the model presented in a particular way.

The idea of having separately the logical model and the views is especially important in the CASE tools that can process models. For example, IDK [8,15] generates the programming code from a MAS model.

A same element can take place in two different diagrams. In fact, this element is supposed to be a single element. For a CASE tool, if the model was just the set of diagrams, it would be difficult to process the elements that take place in several diagrams. For instance, in a CASE tool, when an element changes in one diagram, it should change automatically in the other diagram. In addition, when generating code, all elements of a model should be considered together and once.

In conclusion, the presented technique defines the diagrams as views of the model. An instance of a modeling language metamodel root contains all the entities and relationships of M1 layer. This instance is denoted as *dictionary*. In the dictionary, each element (entity or relationship) is represented only once. An instance of a diagram definitions metamodel has diagrams of M1 layer. These diagrams do not contain the elements. Instead, these diagrams point to the elements of the dictionary. A single element in the dictionary can be referenced by several diagrams. Thus, the diagrams are not containers of elements but views of the dictionary.

4.5 The Diagram Definitions Metamodel with the ECore Language

This section describes how to represent the *diagram definitions metamodel* with the *ECore* language. The metamodels generated by DTET use this representation.

As mentioned in Sect. 4.4, the goal is to define diagrams as views of the dictionary of elements. The key is to properly combine the *containment* and *non-containment* references. These two kinds of references were described in Sect. 3. The dictionary has containment references. Then, the dictionary contains the entities and relationships. The diagrams have non-containment references to the entities and relationships of the dictionary. Thus, the diagrams contain neither the entities nor relationships. The diagrams just have references to the elements at the dictionary.

In particular, DTET creates diagram definitions metamodels with a specific structure. Let `<name>` be the diagram name defined by the user. Then, a diagram definition is represented with the following elements:

- A class called `<name>Diagram` for the diagram body. This class is placed in a package called *diagrams*. The class has two containment references:
  - The `<name>DE` reference. It references a class called `<Name>DE`, which is described below.
  - The `<name>DR` reference. It references a class called `<Name>DR`, which is also described below.
- A class called `<Name>DE`. It has a multiple non-containment reference for each entity included in the diagram called `<name>`. The entities are defined in the modeling language metamodel. The `<Name>DE` class is situated in a package called *de*.
- A class called `<Name>RE`. It has a multiple non-containment reference for each relationship included in the diagram called `<name>`. This class is similar to `<Name>DE`. The difference is that this class references relationships instead of entities. The `<Name>RE` class is placed in a package called *dr*.

In this manner, the diagrams keep separately the references to the entities and the references to the relationships.

Figure 6 shows an example of a excerpt of particular diagram-type definition within a metamodel, following the current approach. Specifically in this example, the metamodel defines a diagram type called *Interaction Model* within the Ingenias language. This language is aimed at modeling MASs, and this diagram type determines the interactions among agents. In particular, this diagram type includes the *InteractionUnit, Task, Role* and *Agent* entities. The agents are the autonomous entities within the system, and the roles represent pattern behaviors. An interaction is composed of several interaction units, which represent the messages interchanged among different agents. The tasks are executed as result of receiving or sending a specific message. The *IInitiates, IColaborates* and *UIPrecedes* relationships are included in this diagram type. IInitiates associates an agent that sends a specific message executing a particular task, while IColaborates does the same but for the reception of a message. Finally, UIPrecedes determines the order in which the interaction units are interchanged in the interaction.

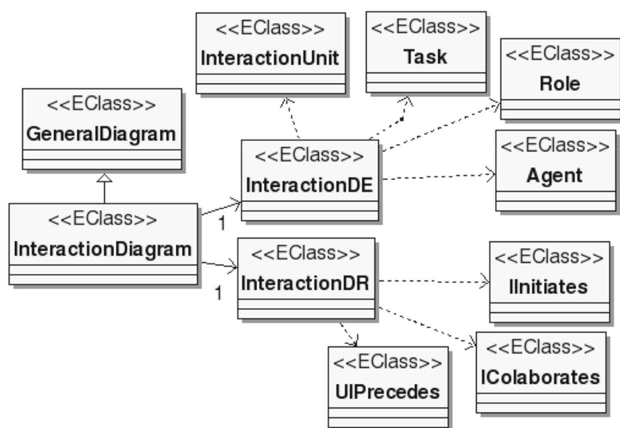

Fig. 6 The *Interaction* diagram type of the Ingenias language, with continuous arrows for containment references and discontinuous arrows for non-containment references

Besides the aforementioned metamodeling elements, the diagram definitions metamodel needs to have other elements:

- A class called *GeneralDiagram* (see Fig. 6). All the diagrams must extend this class in the metamodel.
- A class called *ConcreteSpecification* (see Fig. 7). The instance of the ConcreteSpecification class contains the diagrams of the instance models in M1 layer. For this purpose, this class has a containment reference called *diagrams* of *GeneralDiagram* type. This class extends the Specification class.

Finally, there are two possible configurations for the instance models in M1 layer. These configurations are the following:

- *The instance model in M1 layer is represented with only one file*. This file is an instance of the ConcreteSpecification class of the diagram definitions metamodel. The dictionaries are kept in this class by means of the references that are inherited from the Specification superclass. This instance also contains the diagrams.
- *The instance model in M1 layer is represented with two files*. The first file is an instance of the Specification class of the modeling language metamodel. The second file is an instance of the ConcreteSpecification class of the diagram definitions metamodel. In this manner, the dictionary and the diagrams are also kept separately in two files in the M1 layer. In most cases, this work recommends to use this option because it is more modular than the other option.

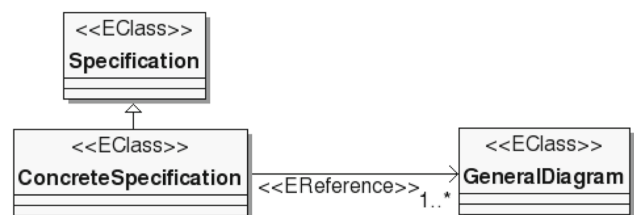

Fig. 7 The concreteSpecification class

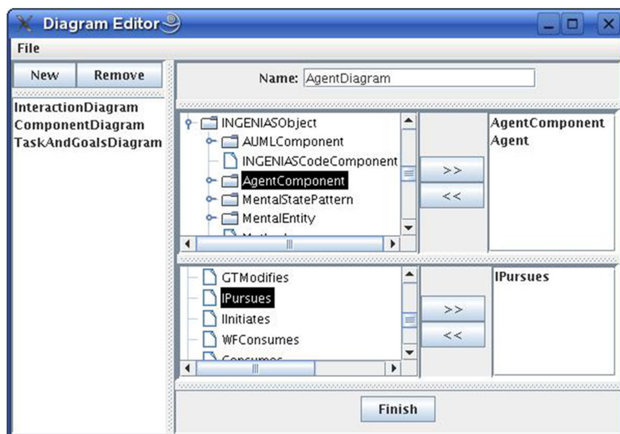

Fig. 8 DTET, the presented tool for metamodeling diagram types

5 DTET

DTET allows users to define diagrams types for most modeling languages, as indicated in Sect. 4.2. Once the users have defined all the diagram types, DTET generates code with the ECore metamodeling language, and this code constitutes a particular diagram definitions metamodel.

The GUI of DTET is presented in Fig. 8. On the left side, the diagrams are listed. Above the diagrams list, there are two buttons with the *New* and *Remove* labels. The user can create a new diagram or remove an existent diagram, respectively, by means of these two buttons.

On the right side of the GUI, which is wider than left side, there is a panel for creating a diagram type. On the top of this panel, there is an edit field for introducing the name of the new diagram. Below this edit field, there are two selection areas, respectively, for entities and relationships. In the first area, the user can select the entities of the modeling language for including them on the new diagram type. The user selects an entity from the tree hierarchy of entities. Then, when pressing the button with label ">>", the selected entity is added to the selected entity list on the right. In the entity hierarchy of the GUI, the background color of the selected entities changes. The goal of this change of color is to inform which entities of the hierarchy are already selected. The button with label "<<" provides the possibility of removing an entity from the selected entities list.

It is worth highlighting that, when adding an entity from the hierarchy to a diagram type, it is not necessary to select any of the descendant entities. The reason is that the hierarchy considers a *Is_A* relationship between an entity and its parent. Thus, when an entity is included, all its descendants are assumed to be included.

The mechanism for selecting relationships for the new diagram type is similar to the entity selection mechanism. The relationships are selected from the relationships hierarchy. When the user finishes defining the whole diagram type, the user presses the *Finish* button. Then, the new diagram type is added to the diagrams list.

Finally, on the *File* menu, there are three options. Firstly, the user can load a modeling language metamodel, and the tool updates the hierarchies of entities and relationships. Secondly, the user can save a list of diagram definitions. DTET creates a diagram definitions metamodel. Thirdly, the user can load a diagram definitions metamodel, which can have been previously created by DTET.

6 Evaluation

DTET has been evaluated by comparing it with other tools, following a quantitative fixed design type with a experiment strategy according to the empirical strategies proposed by Wohlin et al. [33]. This evaluation is aimed at comparing DTET with other alternatives in terms of average time for defining diagram types and average number of errors in these definitions.

In particular, Sect. 6.1 introduces the context of this evaluation, arguing the reasons for selecting certain alternative editors and certain CASE tools as scenarios for the evaluation. Section 6.2 describes the subjects that have been selected for the current experimental evaluation. Section 6.3 mentions the hypothesis of the evaluation, decomposing it in simpler hypotheses that can be proved or refuted. Finally, Sect. 6.4 presents the results of the experiments, and analyzes these with statistical tests and charts.

6.1 Context

To the best of authors' knowledge, there is not any other tool specifically aimed at defining diagram types with metamodels. Therefore, DTET has been compared with general-purpose editors for metamodels. These editors usually depend on the metamodeling language. The most common metamodeling languages are ECore and MOF, considered the high number of metamodels defined in these languages in comparison with the number of metamodels defined in other languages. Bearing in mind those two languages, ECore language has been selected for choosing alternatives because the current technique is also represented with this language. In this manner, the comparison is not affected by the peculiarities of the metamodeling languages. Considering the ECore metamodeling language, the main editors are those from the Eclipse foundation, as this foundation is the one that defined and promoted ECore. Eclipse provides the EMF editor as the basis for defining ECore metamodels. This is the most popular editor for defining ECore metamodels, and subsequently, EMF editor has been selected for the current comparison. ECore metamodels can also be edited

with a text editor, which is one of the most flexible editors for this purpose, but it also needs a lot of expertise in the language. Text editor has been considered flexible and popular enough to be selected as another alternative for this comparison. There are other Eclipse plugins that are related to ECore metamodels, such as the Graphical Modeling Framework (GMF) and EcoreTools. GMF was discarded since it is not flexible enough to define diagram types because of its structure. EcoreTools extends EMF, but it is very similar regarding the definition of ECore metamodels. Thus, only one editor is selected from EMF and EcoreTools. Specifically, EMF editor was selected because it is more popular.

There are numerous CASE tools that define connection-based modeling languages with diagram types. For instance, regarding UML tools, there are plenty of these such as StarUML, BoUML, Dia, ArgoUML, Rational Rose XDE and Microsoft Visio, among others. From these, BoUML [19] has been selected for this comparison because (1) it is one of the most popular tools; (2) it is free, and consequently, it is accessible for a wider audience, including the testers that participated in this experiment; and (3) it is strictly related to one modeling language metamodel, i.e., the UML metamodel [27,28] (e.g., others such as Dia manages more languages than UML). Considering domain-specific tools, there are many domains that use these tools. Among others, examples of these domains are representation of chemical compositions, MASs, databases, processes, telephone nets and electronic circuits. The domain of MASs has been selected since (1) it has many modeling languages (e.g., Prometheus, Ingenias, Adelfe and Gaia) supported with tools with several diagram types, and (2) most of these languages include a great amount of entities and relationships. In particular, this evaluation has chosen the IDK [8] tool supporting the Ingenias language [15], because (1) it is one of the most popular tools in this domain, (2) its language has one of the larger metamodels in this domain, being this useful for comparing definition times with metamodel sizes, and (3) it includes a model-driven approach for automatically generating programming code, which makes it interesting for many practitioners in this domain. Finally, the domain of processes has been selected due to the facts that (1) it supports an standard metamodel, i.e., the Software & Systems Process Engineering Metamodel (SPEM) [29], and (2) it is used by a wide audience, i.e., developers of both software and systems. Among other tools such as Visio and Eclipse SPEM Designer, this evaluation chooses APES2 [17] because (1) it is only related to the SPEM metamodel and not others, (2) it is free and can be accessible by a wider audience, (3) the testers already knew this tool and (4) it uses a short subset of the metamodel, adding this variety to the evaluation when considering different metamodel sizes.

Each of the selected tools uses diagram types, and each of these associated with a certain number of elements. Concretely, Table 1 indicates the number of diagram types (in *Diag.* column), the number of associations between the diagram types and the element types (in *Elem.* column), and the sum of these two columns as the size of the definitions of diagram types (in *Size* column).

Table 1 Features of the diagram types of CASE tools

	Diag.	Elem.	Size (Diag. + Elem.)
IDK	10	157	167
BoUML	6	63	69
APES2	1	5	6
Total	17	225	242

There was a version of the IDK metamodel that already satisfied the requirements established in Sect. 4.2. However, it was necessary to adapt the subset of the UML metamodel for BoUML and the subset of SPEM for APES2. In this manner, DTET was able to support these metamodels of these CASE tools. These preadaptation phases followed the steps determined in Sect. 4.3.

6.2 Selection of Subjects

The current experiments were experienced by 36 students of the Software Architecture Advanced Design subject of the Master Degree in Software Architecture, and three experts in MDE with PhD degree. The background of all these testers have in common to know the principles of MDE and the UML-DI standard and to be used to CASE tools with different diagram types. In addition, all of them had defined several DSMLs before this experiment. These testers were randomly divided into three groups, each of which with an expert with PhD degree and 12 students randomly selected. All the members of the three groups defined all the diagram types of the different CASE tools, using different tools (text editor, EMF editor, DTET). In order to avoid the influence of testers that remember the diagram types of a CASE tool when redefining these diagram types, each group of testers followed a different order. Specifically, the first group followed the order of editors (text, EMF and DTET), the second one (EMF, DTET and text) and the third one (DTET, text and EMF).

Regarding the geographical distribution of testers, 16 students are from Spain, 10 students are from Dominican Republic, 8 students are from Colombia and 4 students are from Ecuador. In addition, two PhD testers are from Spain, and one PhD tester is from France.

6.3 Hypothesis

The hypothesis of this evaluation is that practitioners need less time and make less errors when defining diagram types

Table 2 Hypotheses of the evaluation

Hypothesis	Definition
HT_0	Practitioners need a similar amount of time for defining diagram types with DTET and each alternative
HT_1	Practitioners need different amounts of time for defining diagram types with DTET and each alternative with a significant difference
HE_0	Practitioners make a similar amount of errors for defining diagram types with DTET and each alternative
HE_1	Practitioners make different amounts of errors for defining diagram types with DTET and each alternative with a significant difference

with DTET than with the alternatives with statistically significant differences. In particular, this hypothesis has been converted in two simpler null hypotheses and their alternatives for conducting the statistical analysis. These hypotheses are named as HT for the hypotheses regarding time, and as HE for the hypotheses regarding the numbers of errors. In particular, HT_0 and HE_0 are the null hypotheses, while their alternative hypotheses are called HT_1 and HE_1. Table 2 defines all these hypotheses.

6.4 Analysis of the Results

For all the testers, when defining the diagram types of each tool, the necessary time and the number of errors were measured for each editor. In the case of DTET, the time of the preadaptation phase and the time of using the tool have been measured separately, and the sum of these times is considered in the comparison. The averages of these results are presented in Table 3. The definition of the diagram types of all tools took 6 h and 20 min with the *text editor* in average, and 65.95 errors were encountered in average. The diagram-type definitions took less time (about 2 h and 27 min) and implied fewer errors (i.e., 10.35 errors) with the *EMF editor*. However, the best results are obtained with DTET (only 1 h and 5 min and 0.11 errors). The errors in these experiments were mainly human errors. For example, in the case of the text editor, the errors were mainly related to misspelling and syntax. In the case of the EMF editor, most errors were related to incorrectly set the *containment* and *upper bound* properties of the references. These errors make it impossible the instantiation of models. The errors with DTET were only related with the wrong association of existing elements with diagram types, for instance, due to the similarity of some entity names (e.g., *DeploymentUnitByTypeEnumInitMS* and *DeploymentUnitByTypeMSEntity* in the IDK).

In order to determine whether the differences of the results are statistically significant, this work has applied statistical tests. Firstly, this analysis applies the Shapiro–Wilk test of normality. The data concerning the number of errors in the

Table 3 Comparison of DTET with the text editor and EMF editor considering the time (hh:mm:ss) and the number of errors of the definition of diagram types

	Text editor		EMF editor	
	Time	Errors	Time	Errors
IDK	4:32:07	43.71	1:40:33	7.02
BoUML	1:36:20	20.22	0:42:31	3.33
APES2	0:12:15	2.02	0:04:20	0.00
Total	6:20:42	65.95	2:27:24	10.35
	DTET			
	Pre-adaptation	Tool time	Time	Errors
IDK	0:00:00	0:38:50	0:38:50	0.08
BoUML	0:10:11	0:13:05	0:23:16	0.03
APES2	0:02:30	0:01:10	0:03:40	0.00
Total	0:12:41	0:53:05	1:05:46	0.11

Table 4 Results of the Mann–Whitney tests for time

Null hypothesis	Compared editors	Sig.	Difference of means (s)	Decision
HT_0	Text editor and DTET	0.000	−18,896	Reject the null hypothesis
HT_0	EMF editor and DTET	0.000	−4898	Reject the null hypothesis

EMF editor passed the test of normality (significance 0.372). However, this test of normality failed for the data regarding errors in the other editors and the data regarding time in all the editors (significance below 0.050 in all cases). For this reason, this analysis applies a non-parametric test.

In particular, this work applies the Mann–Whitney test for comparing DTET with each alternative, as this analysis is performed with three independent samples. These tests are applied considering the sum of time and the sum of errors of the three scenarios for each subject. Table 4 presents the results of this test for time of defining diagram types. In this table, *sig.* stands for statistical significance. One can observe that this test rejects the HT_0 null hypothesis for, respectively, the differences between DTET and both alternative editors. Hence, the alternative HT_1 hypothesis is concluded. Differences of time between DTET and each of the alternatives are statistically different (with a significance of 0.000 in both cases). This table also includes the difference of means, which are -18896 seconds between text editor and DTET and −4898 seconds between EMF editor and DTET. The time is less with DTET than both alternatives.

Table 5 presents the results of Mann–Whitney test for the number of errors. One can observe that HE_0 null hypothesis is rejected for both alternative editors. Thus, practitioners make different numbers of errors with DTET and each of

Table 5 Results of the Mann–Whitney tests for errors

Null Hypothesis	Compared Editors	Sig.	Difference of means	Decision
HE_0	Text editor and DTET	0.000	−65.84	Reject the null hypothesis
HE_0	EMF editor and DTET	0.000	−10.24	Reject the null hypothesis

Table 6 Comparative percentages of time and errors

	Time		Errors	
	DTET/text (%)	DTET/EMF (%)	DTET/text (%)	DTET/EMF
IDK	14.27	38.62	0.18	1.14 %
BoUML	24.15	54.72	0.15	0.90 %
APES2	29.93	84.62	0.00	DIV/0
Average	22.79	59.32	0.11	1.02 %

the two alternatives, with statistically significant differences (significance of 0.000 in both alternatives). Thus, the alternative HE_1 hypothesis is confirmed for both editors. The number of errors is less with DTET than each alternative. In particular the differences of means are -65.86 errors between text editor and DTET, and -10.24 errors between EMF editor and DTET.

The ratios of times between DTET and each of the alternatives have been calculated for each CASE tool scenario. Table 6 presents these ratios. This table also includes the ratios of the number of errors with DTET and each of the alternatives. In average, one can observe that with DTET practitioners only needed the 22.79 % of the time that they needed with the text editor and the 59.32 % of the time with EMF editor. Regarding the errors, the ratios were considerably lower. Specifically, the number of errors with DTET was only 0.11 % of the errors with the text editor and 1.02 % of the errors with EMF editor. Therefore, practitioners spent less time for defining types with DTET than the considered alternatives, and they made much less errors in this activity.

The data of the experiments are presented graphically separating the data about time from the data about errors. In particular, Fig. 9 presents a comparison of the average times of defining the diagram types of each CASE tool with the DTET and the alternatives. As one can observe, practitioners needed less time with the current approach (considering the sum of preadaptation phase and definition with DTET) than with the alternatives for each considered CASE tool. Moreover, Fig. 10 presents the percentage of average times spent with DTET divided by each alternative. These percentages are presented in a dispersion chart considering the sizes of diagram types

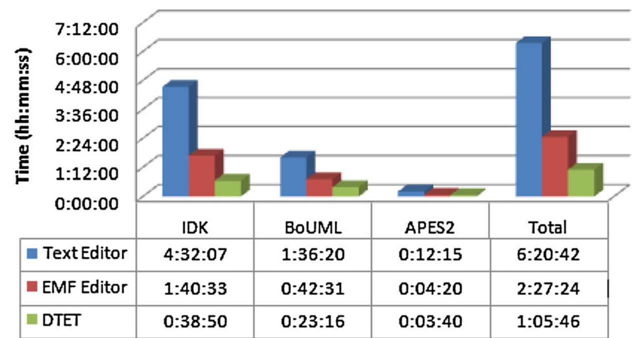

Fig. 9 Comparison of times for defining diagram types

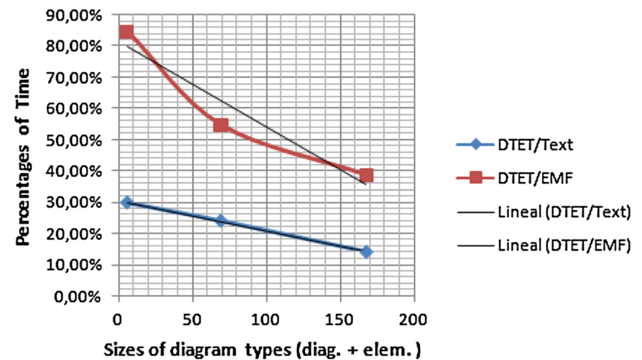

Fig. 10 Comparison of time percentages regarding the sizes of diagram types

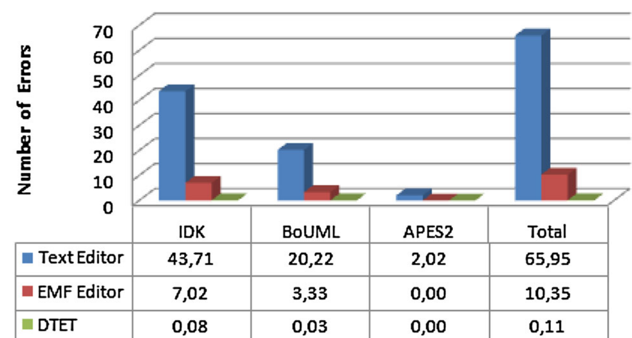

Fig. 11 Comparison of errors when defining diagram types

in the abscissas. One can observe that the percentage of time of using the current approach with DTET in comparison with alternatives decreases when modeling larger diagram types. Thus, the current approach is more efficient in relation with the analyzed alternatives when the diagram types are larger.

Regarding the errors, Fig. 11 shows the errors of alternatives and DTET, and one can observe that DTET has much fewer errors than alternatives. In particular, Fig. 12 shows the percentages of errors regarding the sizes of definitions of diagram types. As illustrated with this figure, DTET has less than 1.20 % of errors for any tool and alternative considered.

In summary, the definition of the diagram types takes less time with DTET than with the other alternatives, and this

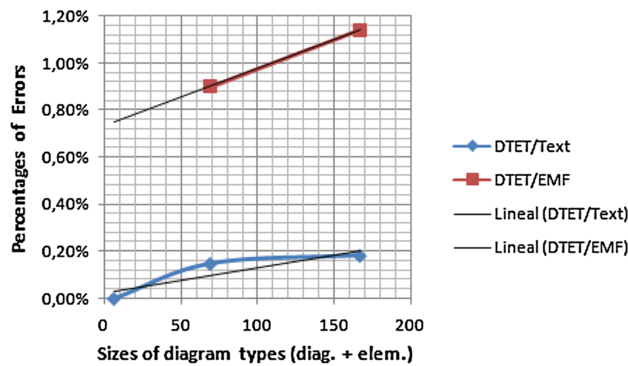

Fig. 12 Comparison of percentages of errors regarding the sizes of diagram types

difference becomes more notorious in larger definitions of diagram types. In addition, DTET barely let users make mistakes. Making mistakes is more probable with the other considered alternatives.

7 Discussion About Related Works

This section compares the current work with other similar approaches. Firstly, Sect. 7.1 discusses the advantages and downsides of this approach in comparison with the UML-DI approach, and Sect. 7.2 analyzes other related works. The current approach is compared both quantitatively and qualitatively, following the recommendations of Wohlin et al. [33]. In particular, the previous section quantitatively compared DTET with other alternative tools that can define diagram types. Section 7.3 now compares DTET with the alternatives from a qualitative point of view, mentioning both advantages and drawbacks of DTET. It also discusses some features of the underlying technique.

7.1 Comparison with the UML-DI Approach

The UML-DI specification [4] provides the definition of the UML diagrams. In UML-DI, a *Diagram* may be used to show an extract of an existing *Diagram* without the need of creating the used *DiagramElements* more often than once. For example, a large *Diagram* may be shown through several smaller *Diagrams* for a better representation. To achieve this, the large *Diagram* would own all its containing *DiagramElements* while the *cutout Diagrams* would only contain *References* to the original *DiagramElements*. *References* and stand-alone *DiagramElements* may be used independently in the same diagram.

The UML-DI approach has some disadvantages:

- *Processing of models is difficult in some cases*. Each CASE tool designer must decide what to do with the cutouts of diagrams. If the cutouts are processed, a same element may be processed twice. If not, maybe some relationships are missed. It is possible to process the models but a bit confusing. Some CASE tools, such as BoUML [19] and Rational Rose [30], use a different internal representation from the model representation to solve the mentioned problem. However, the experts on DSMLs [**?**] discourage this practice.
- *There can be some undetermined situations*. Maybe a diagram has a cutout of another diagram. In this cutout of the diagram, there can be another cutout of another diagram. In this case, each CASE tool designer must decide whether, in the first diagram, the cutout is taken from the second or the third diagram. This decision is relevant because, if the user removes the second diagram, what happens with the cutout of the first diagram is not determined. Implementing all these situations in a CASE tool is quite expensive compared to the approach presented in this paper.

On the contrary, the current approach defines models with a single container (dictionary) of all the elements (entities and relationships), and there are several views (diagrams) of this single container. In this manner, in the proposed approach, the processing of models usually only requires processing the dictionary instead of processing the diagrams, the cutouts and their dependencies. Moreover, in the present approach, each diagram directly references the elements in the dictionary avoiding any dependency with any other diagram, and consequently preventing diagrams from undetermined situations.

Furthermore, the present approach can be applied to a large variety of modeling languages and can define new diagram types. By contrast, UML-DI can be only applied to UML and does not consider defining new diagram types.

Nevertheless, UML-DI describes useful information that this paper does not. For example, the inclusion of spatial information is not described in this paper. The description of this information is left for future work.

7.2 Discussion of Other Related Works

To begin with, in 2002, Kalnins et al. [22] introduce the principles of a generic modeling tool. This work uses the *diagram patterns*. The diagram patterns are not the same as diagram types, but they have similarities. These diagram patterns are common constructions in the modeling diagrams. By contrast, diagram types establish certain restrictions on what entities and relationships a diagram type can contain.

Related to our work, Celms et al. [20,24] present a technique for defining diagram types. However, their approach uses general-purpose tools for the definition of diagram types. Since the current work is based on the specific tool

Table 7 Qualitative comparison of DTET with alternatives

	Text Editor	EMF Editor	DTET
Possible Errors			
Spelling errors in the ECore keywords	Yes	No	No
Errors in the XML syntax	Yes	No	No
Conceptual errors in the structure of diagram definitions metamodels	Yes	Yes	No
Spelling errors in the domain-specific entities and relations	Yes	Yes	No
Wrong associations of elements with diagram types according to the domain	Yes	Yes	Yes
Features			
GUI specifically aimed at defined diagram types	No	No	Yes
Friendly way of setting associations between diagram types and elements	No	No	Yes
Required expertise from users	High	High	Low
Required preadaptation phase	No	Only translation to ECore	Yes
Inclusion of new elements in the modeling language	Yes	Yes	No

support for the diagram-type definition, the definition of diagram types is faster and safer from human errors with the current technique than with their technique. In addition, their work is based on the definition of mappings for the definition of diagram types. Conversely, the present technique defines a metamodel for defining the diagram types. One of the main advantages is that the use of the metamodel makes the instantiation of diagrams clear.

Moreover, the definition of diagrams types can be useful in several contexts. To begin with, Koskinen et al. [23] propose operations to compare, merge, slice and synthesize UML diagrams. These operations are based on the UML diagram types. In addition, Boronat et al. [18] provide the merge operation for class diagram integration. When considering diagram types, the formal diagram-type definitions can be useful. Furthermore, Alanen et al. [16] discuss how to create and update diagrams after model transformations. This work uses the diagram definitions. Thus, diagram-type definitions can be important when creating or updating diagrams after model transformations. Heuer et al. [21] determine a mechanism for assessing the variability in two different diagram types, which are UML activity diagrams and petri nets. A normalized way of formalizing different diagram types, as the current work proposes, can be useful for works that present a common research in different diagram types.

Finally, in the metamodeling literature, there are several works that define several diagram types. For instance, Wagner [32] presents a metamodel for MASs with several diagram types. In addition, Vroom [31] presents an example of industrial design engineering by means of a metamodel that has six diagram types. Moreover, Molina et al. [26] define a DSML for specifying interactive groupware applications with a metamodel with ECore. This DSML has several diagram types and is supported with a tool for designing models and another one for validating these models. Laleau and Polack [25] define new diagram types for Information Systems (IS) departing from UML diagrams, conforming the new modeling language called IS-UML. In particular, they define IS transactions with a new diagram type that is the combination of two types of UML diagrams (collaboration and state diagrams). They also introduce a new diagram type that extends the UML class diagrams. However, none of these works proposes a technique for metamodeling diagram types.

7.3 Qualitative Comparison

This section compares DTET with the considered alternatives (i.e., text editor and EMF editor) from a qualitative point of view, with a flexible design type based on the case study strategy of Wohlin et al. [33]. This qualitative comparison is summarized in Table 7. One of the main advantages of DTET is that users can only make a possible kind of mistake, which is that they wrongly associates an existing element of the language with a diagram type, according to the domain. For instance, this may occur if there are several elements of the language that have similar names.

The mistake of misspelling domain-specific entities and relations is not possible with DTET, since the user selects these entities and relations from two lists of choices. However, this is possible with the text editor, and with the EMF editor if the references to external files are not explicitly checked.

DTET does not allow users to define diagram definitions metamodels with wrong structures, as it manages this struc-

ture. However, these errors are possible in the alternatives. Finally, there are some errors that are only possible in the text editor, such us spelling errors in ECore keywords and syntax errors of the XML structure.

DTET has other advantages over the alternatives such as a GUI specifically designed for defining diagram types, and its friendly way of setting associations between diagram types and elements (i.e., by selecting elements from lists of choices). Another advantage is that the required metamodeling expertise of users can be low in comparison with the expertise level necessary for defining diagram types with the alternatives.

Nonetheless, DTET has some few disadvantages in comparison with the alternatives. Firstly, DTET usually needs a preadaptation phase to adjust the metamodel of the modeling language, so that DTET can process it and allow users to define diagram types. This is not necessary in the text editor. In the case of the EMF editor, it is only necessary to translate the metamodel to ECore, only if it is represented with a different language. Another downside of DTET is that it only allows users to define the diagram definitions metamodels and sometimes users can miss the possibility of changing the modeling language metamodels within the same tool. Oppositely, this is allowed in the alternatives.

Regarding the underlying technique of the current approach, this paper has already mentioned the main advantages of the current approach over other options, and these advantages are omitted here for the sake of brevity. However, this section discusses some drawbacks that are not mentioned before so that researchers can have a complete view of the present approach. In particular, the use of a unique dictionary for all the diagrams may imply some negative features, especially in distributed modeling. For instance, the unique dictionary is a single point of failure. In other words, if the dictionary fails, the modeling of all diagrams would stop working. In the case of distributed modeling, this single-point-failure problem can be notorious, since it is possible that the device with the dictionary stops working while the other devices continue running. In this case, the users of the other devices would suffer the failure.

Another disadvantage of the unique dictionary is that it can become a performance bottleneck when there are many diagrams. This performance bottleneck can be especially noticed if several diagrams are simultaneously changed, for example, in distributed modeling applications.

8 Conclusions and Future Work

A technique is presented for defining diagram types of modeling languages by means of metamodels. This technique mainly covers the gap of the literature of non-UML connection-based DSMLs, which is not supported by its alternative standard (i.e., UML-DI). It also has some advantages for CASE tools of UML over UML-DI. The present technique is supported by a tool called DTET. This tool receives input from a metamodel of a modeling language and provides a GUI in which users can define diagram types. Then, DTET generates a metamodel that defines the diagram types. The modeling language and diagram types are separately kept in different metamodels. Some CASE tools can be generated from these metamodels. The models that are instantiated with this technique can be processed more easily than in other approaches, and some undetermined situations are avoided. DTET is experimentally shown to be faster for defining diagram types than the alternative metamodel editors. This difference is statistically significant and is more notorious for larger definitions of diagram types. Experimentation also shows that DTET is less error-prone for defining diagram types than the alternative editors with statistically significant differences. Finally, DTET has also been compared with the alternatives from a qualitative point of view, analyzing its advantages and its downsides.

DTET is planned to be integrated in Eclipse as a plugin in the future. In particular, this plugin will create DSML projects that will be observed in the *projects tree* window, and each of these will be able to contain a modeling language metamodel and a diagram definitions metamodel. The second metamodel will be able to be edited with either the DTET or the EMF editor, both of which will be able to be displayed in the central window of Eclipse. In this manner, within an Eclipse environment, designers can easily create metamodels for diagram types with a customized tool and open them with the EMF editor. The toolbar and pop-up menus will include options for creating a diagram definitions metamodel, saving it, and modifying an existing one.

Furthermore, the presented technique and DTET cannot define certain aspects concerning diagrams. Firstly, it cannot define nested diagrams, which are diagrams that are included in other diagrams. Secondly, certain constraints involving groups of elements cannot be defined only with DTET. For instance, this could be necessary for defining constraints for supporting code generation in some DSMLs. The solutions for these problems are planned to be further considered in the future in the proposed technique and DTET. For instance, nested diagrams can be defined by using diagram elements that can contain references to other diagrams. The additional constraints can be defined by means of the Object Constraint Language (OCL), and these constraints will be able to be introduced from DTET.

Finally, this work can be improved with an explicit description of the spatial information in diagrams. This improvement will probably alter neither the presented technique nor DTET, because the spatial information is planned to be included in a separate metamodel.

References

1. Tayan, O.; Al BinAli, A.M.; Kabir, M.N.: Analytical and computer modelling of transportation systems for traffic bottleneck resolution: a Hajj case study. Arab. J. Sci. Eng. **39**(10), 7013–7037 (2014)
2. Hassan, A.; Kassem, A.M.: Modeling, simulation and performance improvements of a PMSM based on functional model predictive control. Arab. J. Sci. Eng. **38**(11), 3071–3079 (2013)
3. Memon, R.N.; Salim, S.S.; Ahmad, R.: Analysis and classification of problems associated with requirements engineering education: Towards an integrated view. Arab. J. Sci. Eng. **39**(3), 1923–1935 (2014)
4. OMG:UML Diagram Interchange Specification. Tech. rep., Object Management Group, http://www.omg.org/ (Last Accessed 18 Dec 2014), (2006-04-04)
5. Rasool, G.; Mäder, P.: A customizable approach to design patterns recognition based on feature types. Arab. J. Sci. Eng. **39**(12), 8851–8873 (2014)
6. Alshayeb, M.; Eisa, Y.; Ahmed, M.A.: Object-oriented class stability prediction: a comparison between artificial neural network and support vector machine. Arab. J. Sci. Eng. **39**(11), 7865–7876 (2014)
7. Costagliola, G.; Delucia, A.; Orefice, S.; Polese, G.: A classification framework to support the design of visual languages. J. Vis. Lang. Comput. **13**(6), 573–600 (2002)
8. Ingenias.: INGENIAS Development Kit. http://ingenias.sourceforge.net/ (Last Accessed 18 Dec 2014), (2014)
9. OMG.: Meta Object Facility (MOF) Specification. Version 1.4. Tech. rep., Object Management Group, http://www.omg.org/ (Last Accessed 18 Dec 2014), (2002-04-03)
10. Budinsky, F.: Eclipse Modelling Framework: Developer's Guide. Addison Wesley, Boston (2003)
11. Moore, B.; Dean, D.; Gerber, A.; Wagenknecht, G.; Vanderheyden, P.: Eclipse Development Using Graphical Editing Framework and the Eclipse Modelling Framework. IBM Redbooks, New York, (2004)
12. García-Magariño, I.; Fuentes-Fernández, F.; Gómez-Sanz, J.J.: Guideline for the definition of EMF metamodels using an entity-relationship approach. Inform. Softw. Technol. **51**(8), 1217–1230 (2009). doi:10.1016/j.infsof.2009.02.003
13. Kelly S.: GOPRR Description. PhD dissertation Appendix 1 (1997)
14. García-Magariño, I.; Fuentes-Fernández, R.: A technique for defining metamodel translations. IEICE Trans. Inform. Syst. **92**(10), 2043–2052 (2009)
15. Pavón, J.; Gómez-Sanz, J.J.: Agent oriented software engineering with INGENIAS. Multi-Agent Syst. Appl. III **2691**, 394–403 (2003)
16. Alanen, M.; Lundkvist, T.; Porres, I.: Creating and reconciling diagrams after executing model transformations. Sci. Comput. Progr. **68**(3), 128–151 (2007)
17. Apes: APES2: A Process Engineering Software. http://apes2.berlios.de/en/links.html (last accessed 12/18/2014), (2014)
18. Boronat, A.; Carsí, J.; Ramos, I.; Letelier, P.: Formal model merging applied to class diagram integration. Electron. Notes Theor. Comput. Sci. **166**, 5–26 (2007)
19. Bouml.: BOUML: An UML Tool Box. www.bouml.fr (last Accessed 18 Dec 2014), (2014)
20. Celms, E.; Kalnins, A.; Lace, L.: Diagram definition facilities based on metamodel mappings. In: Proceedings of the 18th International Conference, OOPSLA, pp. 23–32 (2003)
21. Heuer, A.; Stricker, V.; Budnik, C.J.; Konrad, S.; Lauenroth, K.; Pohl, K.: Defining variability in activity diagrams and petri nets. Sci. Comput. Progr. **78**(12), 2414–2432 (2012)
22. Kalnins, A.; Barzdins, J.; Celms, E.; Lace, L.; Opmanis, M.; Podnieks, K.; Zarins, A.: The first step towards generic modelling tool. Proc. Baltic DB&IS **2**, 167–180 (2002)
23. Koskinen, J.; Peltonen, J.; Selonen P.; Systä, T.; Koskimies, K.: Model processing tools in UML. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE-01), pp. 819–820 (2001)
24. Lace, L.; Celms, E.; Kalnins, A.: Diagram definition facilities in a generic modeling tool. In: Proceedings of International Conference Modelling and Simulation of Business Systems, Vilnius, pp. 220–224 (2003)
25. Laleau, R.; Polack, F.: Using formal metamodels to check consistency of functional views in information systems specification. Inform. Softw. Technol. **50**(7), 797–814 (2008)
26. Molina, A.I.; Gallardo, J.; Redondo, M.A.; Ortega, M.; Giraldo, W.J.: Metamodel-driven definition of a visual modeling language for specifying interactive groupware applications: an empirical study. J. Syst. Softw. **86**(7), 1772–1789 (2013)
27. OMG.: Unified Modeling Language: Superstructure. Tech. rep., Object Management Group, http://www.omg.org/ (Last Accessed 18 Dec 2014), (2007-11-02)
28. OMG.:Unified Modeling Language: Infrastructure. Tech. rep., Object Management Group, http://www.omg.org/ (Last Accessed 18 Dec 2014), (2007-11-04)
29. OMG: Software & Systems Process Engineering Metamodel Specification (SPEM) Version 2.0. Tech. rep., Object Management Group, http://www.omg.org/ (Last Accessed 18 Dec 2014), (2008-04-01)
30. Rational-Rose.: IBM Rational Software: Rational Rose Family. http://www-03.ibm.com/software/products/en/ratirosefami/ (Last Accessed 18 Dec 2014), (2014)
31. Vroom, R.: A general example model for automotive suppliers of the development process and its related information. Comput. Indust. **31**(3), 255–280 (1996)
32. Wagner, G.: The agent–object-relationship metamodel: towards a unified view of state and behavior. Inform. Syst. **28**(5), 475–504 (2003)
33. Wohlin, C.; Runeson, P.; Höst, M.; Ohlsson, M.C.; Regnell, B.; Wesslén, A.: Experimentation in software engineering. Springer, Berlin (2012)