A Reconciliation Framework to Support Cooperative Work with DSM

Catalin Constantin, Vincent Englebert, and Philippe Thiran

PReCISE Research Centre, University of Namur
Rue Grandgagnage 21, B-5000 Namur, Belgium
{catalin.constantin,vincent.englebert,philippe.thiran}@fundp.ac.be

Abstract. Despite the fact that DSM tools become very powerful and more frequently used, the support for their cooperation has not reached its full strength and the demand for model management is growing. In cooperative work, the decision agents are semi-autonomous and therefore a solution for reconciliating DSM after a concurrent evolution is needed. Many GPML environments are proposed by the CSCW community but they do not usually provide reconciliation, merging or versioning functionalities after asynchronous evolutions. In this paper we propose a reconciliation framework for cooperative work with a DSM language. The main goal is to support the efficiency of software developers and the reuse of software artifacts.

Keywords: CSCW, DSM, metaCASE, model versioning, model reconciliation and merging

1 Introduction

Domain Specific Modelling (DSM) languages are now considered as an efficient alternative to General Purpose Modelling Languages (e.g., UML, Petri Nets) for modelling complex systems [12]. Nevertheless, they require ad-hoc environment tools (called metaCASE tools) that enable method engineers to edit and manage models as well as metamodels. Since the 90's, several tools have been developed such as MetaEdit+ [11], EMF [7], GME [14], and Atom3 [6].

These tools generally consider the modelling task as a single user concern. However, the design of software systems often implies many competences (e.g., middleware engineers, human interface designers, database experts, business analysts). Hence, there is a need to share the modelling artifacts between several engineers and to synchronize their activities. To the best of our knowledge, there is a small number of frameworks that are representative for the cooperative work. MetaEdit+ is a tool that provides a concurrent access to its repository with a *Smart Model Access Restricting Technology* (Smart Locks©). This mode solves the concurrent access to a shared repository but assumes that the information is managed consensually by users. It also offers import-export facilities for managing modelling development. There are other ways of cooperative work[1]: Schmidt

[1] We define "*Cooperative work*" as "*[...] multiple persons working together to produce a product or service.*" from [21].

and al. [21] argue that the group of people concerned by a cooperative task can be large, transient, not stable or even non determinable. They also show that the pattern of interactions can change dynamically and that agents are semi-autonomous in their partial work. There is thus a space for a weakly coupled mode of cooperation where users would have the control of the shared data in an asynchronous way. They could even distribute them without the control of an omniscient central authority. This mode should allow users to have the control of their data, to work in isolation of any other user or a central authority. Implementing the exchange of method chunks [19] could serve as a basis for this kind of collaborative work. In this paper, we consider a *chunk* as a cohesive, autonomous and interoperable model. Standard formats like CDIF [8], GXL [10], PNML [18], or XMI [17] were designed to facilitate the exchange of models among users. Unfortunately, if they define quite well the structure of data, they disregard their semantics. This has jeopardized the interoperability of CASE tools. But the exchange of models between users of the same family of CASE tools is still possible.

We propose to support a weakly coupled cooperation work for DSM users. This mode of cooperation implies dealing with its specific features:

1. Users store models in local repositories;
2. Models can be exchanged between users for long periods of time;
3. Users can be distributed in the world and belong to heterogeneous profiles;
4. Users can work concurrently on models in asynchronous way;
5. Users can delegate their models ownership to other users;
6. When a user owns several models that had a common ancestor model in the past (i.e. they are the result of concurrent modifications applied on a common model — the ancestor), an *oracle* should be able, if possible, to reconcile the models and to merge them into one unique model by preserving the consistency.

Dealing with cooperative issues for DSM tools entails new problems. As DSMs are defined to match as close as possible the application domain, DSMs have to follow its evolution and the new requirements of the stakeholders [23]. MetaCASE tools must then be able to support the evolution of DSM languages during all the project life-cycle. From this observation and from hypothesis 2,4 and 5, we deduce that not only models evolve concurrently, but also their metamodels. Hence, the *oracle* should be able to reconcile both models and metamodels.

In this paper, we argue that a weakly coupled cooperative work is possible for DSM tool users, even if we take into consideration the natural evolution of the language (its metamodel). We present in the next sections the principles of a framework that makes possible the definition of a reconciliation oracle for both models and their metamodels, and that meets the hypothesis of the weakly coupled cooperation model.

The paper is structured as follows: Section 2 gives a small summary of the related work on reconciliation after cooperative work, Section 3 illustrates the issues of cooperative work for DSM tools with an example, Section 4 introduces

the data description language that defines the methods data (e.g., models, metamodels), and the reconciliation framework is introduced in Section 5. Section 6 presents the benefits of our framework and Section 7 concludes the paper.

2 Related Work

Many GPML environments are proposed by the Computer Supported Cooperative Work (CSCW) community but they do not usually provide reconciliation, merging or versioning functionalities for asynchronous evolution. For instance, [15] proposes a cooperative tool supporting distributed edition of UML models but in synchronous mode only.

Regarding DSM, to the best of our knowledge, few approaches have been proposed so far. MetaEdit+ uses a Client/Server approach with a smart locking strategy to share data between users. It also offers import-export facilities for the asynchronous modelling development. This can update the source repository, so that imports into target repositories update the source models there rather than creating new ones. The tool is able to support asynchronous cooperative work with its model patches for common scenarios that are propagated to all users. It allows the users to reference, reuse or create their representation of these core models. Eclipse Modelling Framework Project (EMF) [7] is a modelling framework and a code generation facility for building tools and other applications based on a structured data model. EMF provides resources for developing and implementing UML models using semantic and notational data that is implemented in terms of metamodels defined in EMF. The UML models are partitioned to facilitate a concurrent development and to reduce the likelihood of a non trivial merging. Every change made at the model level generates multiple changes at the metamodel level. The delta (difference) engine is implemented at the EMF level and generates the differences related to both EMF and UML models.

The approach in [20] proposes the use of a versioning system between method chunks. It does not refer to the support of cooperative working and versioning and it is mainly used to solve consistency problems between models and metamodels when a tool allows their independent evolution. [3] describes a proposal for the consistency-receiving merge of model versions, a formalism is introduced to describe the evolution of model revisions which includes a semantic definition of optimistic merge procedure. It however refers the metamodels as static artifacts; only the models are dynamic assets.

Computer Aided Design (CAD) is also a well established field for CSCW as a domain of inquiry. Objects can be described from different viewpoints, representing results of either synchronous or asynchronous cooperative work. Maintaining the relationships amongst these models has been discussed in [13] but CAD tools generally suppose that objects are organized hierarchically and, hence, aggregates are necessarily disjoint [5]. This last hypothesis is too strong in software engineering.

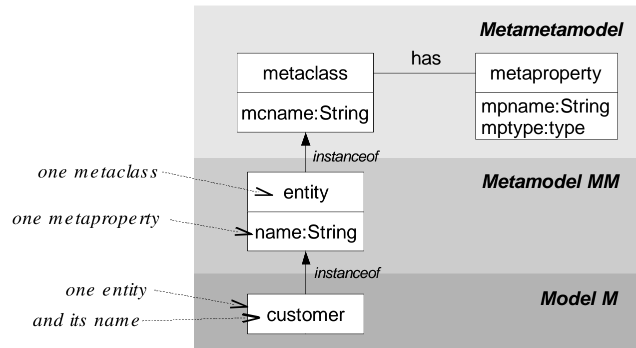

Fig. 1. The Entity-Relationship Model M and its MetaModel MM. The `entity` metaclass owns one metaproperty (`name`) whose type is `String`; `customer` is such entity. The static diagrams are represented with the UML notation for the sake of simplicity.

3 Cooperation Scenario

The cooperation scenario can be illustrated by a simple example. A user *Alice* creates an Entity-Relationship model M along with its metamodel MM (Figure 1). M and MM are then sent to *Bob* to be refined. Bob has then to import model M and its metamodel MM while preserving its local knowledge to avoid any clash between Alice's concepts and his own. Once the data is imported, he extends MM by adding a *persistent* field to the `entity` metaclass, and valuates that field as "SQL" for the `customer` entity. He then renames this entity as "Customer" (with an uppercase), and he finally changes the name of the `entity` metaclass to "EntityType". The result (M_1, MM_1) is sent back to Alice and Bob deletes this information from his repository. The problem consists in merging M with M_1 along with the merging of MM with MM_1. If in the same time, Alice modifies M (i.e. M_2) or MM (i.e. MM_2), the problem becomes more complex: the artifacts evolve independently and the reconciliation of $M1/M2$ and $MM1/MM2$ may require to solve inconsistencies, redundancies, semantic mismatches, etc. For instance, Alice can change the name of the `customer` entity in M from "customer" to "CUSTOMER". This scenario is illustrated in Figure 2. Of course, this scenario must be generalized since models can be the object of much more complex workflows. For the sake of simplicity, only the simple scenario is discussed in this paper.

4 The MetaL Language

This research is carried out in the context of the MetaDone project that aims to develop a metaCASE tool whose functionalities are driven by first-class models. This goal leads us to adopt a fully reified and bootstrapped repository with a specific meta data description language: *MetaL*. Contrary to EMF or UML, there is no distinction between models and metamodels, items can have several types

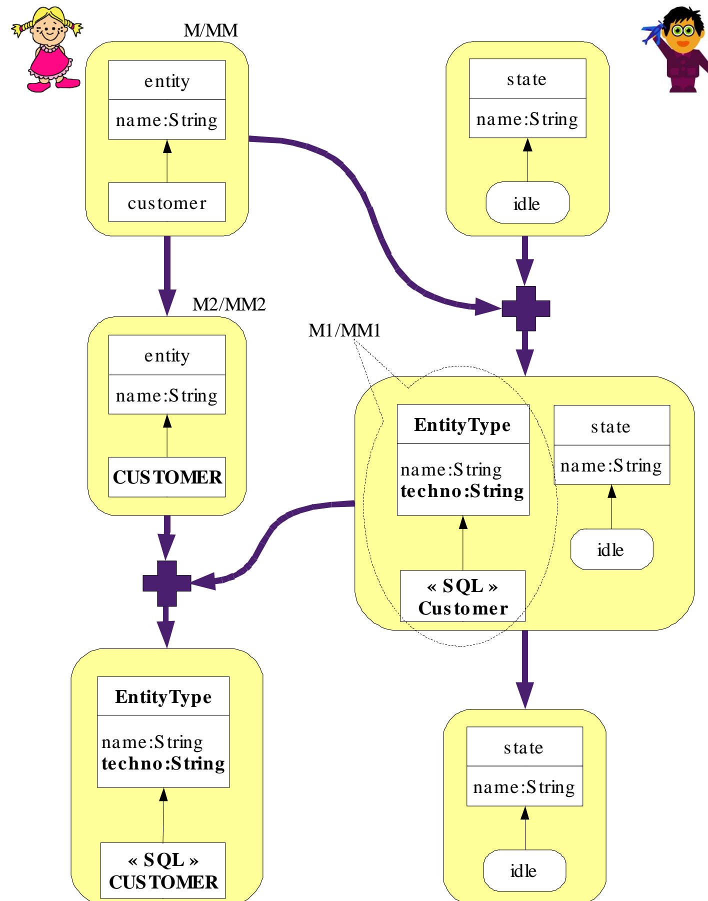

Fig. 2. Cooperative Scenario. This figure shows the exchange of information and the successive evolutions of the (meta)models. Modifications are indicated with bold letters. Cross symbols denote import and reconciliation processes. The upper cross consists mainly in importing the chunk by preserving the local data. The bottom cross requires a more complex reconciliation process.

(e.g. both an EClass and a EAttribute), items can belong to distinct abstraction levels at the same time. The benefits of this approach are discussed in [9]. The next paragraph gives details of the main features of MetaL.

Data is modelled as a set of items D defined as the non-disjoined union of two sets O and P. Elements of O denote objects, while those of P denote properties characterized by a domain and a range that are objects themselves. A subset of O (OT) denotes object types and a many-to-many relationship iof exists between O and OT. In the same way, a subset of P (PT) represents property types and a one-to-many relationship pof is defined between P and PT — a property has exactly one property type. For the sake of simplicity, we will not describe the specialization relationship, and the constraints of this language. A subset OV_S of O will hold string values. Let's just make clear that cycles are allowed in the iof and pof graphs and items can play several roles (object or/and property) at the same time, and can lose or gain roles. A function $name = OT \cup PT \rightarrow String$ maps types to unique identifiers. This language is presented in [9].

Fig. 3. Graphical Syntax of MetaL. Objects and object types are depicted with rectangles while properties and property types are represented with ellipses. The shapes are labeled as "lhs:rhs" where lhs is the facultative name of the item and rhs its type(s) — lhs is sometimes omitted since only types are named. When an item is both a property and an object, its shapes are juxtaposed (this case is not illustrated). Values associated with elements of OV_S are placed on the left side.

Figure 3 illustrates the graphical syntax of the MetaL language and how the DSM language from Figure 1 is described with MetaL. We will reuse this example in the rest of this paper. iof and pof relationships are not represented graphically, but are explicit in our approach.

MetaL describes the first layer of our metaCASE architecture, a second layer (MetaL$_2$) exists with more abstract concepts (metaobjects, metaproperties, metaroles, metamodels, identifiers, cardinalities, ...) in the same way that OWL is defined on top of RDF [22]. Figure 4 illustrates how a DSL statechart can be modelled and used with this language — only an excerpt of MetaL$_2$ is presented. The main important aspect is that all the abstraction levels (e.g., models, metamodels, metametamodels) are stored together by keeping an explicit relationship between "instances" and "types". MetaL is defined to overcome the usual limitations of current DSM languages: limited number of abstraction levels, forbidden relationships between elements of distinct abstraction levels, and management of all the abstraction levels in a homogeneous way. Figure 5 illustrates the use of this modelling language in the context of the MetaDone metaCASE environment.

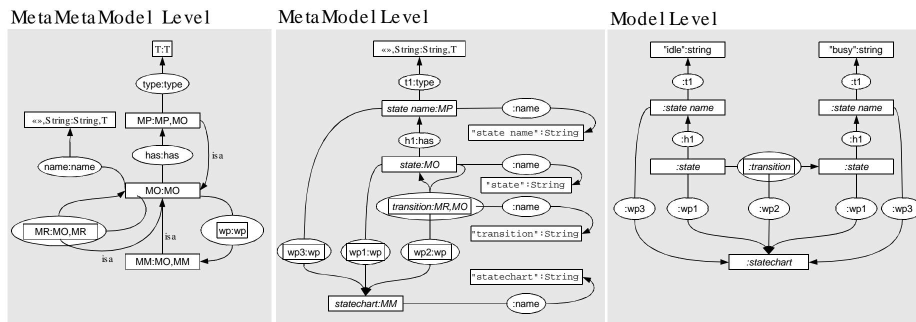

Fig. 4. MetaL$_2$. The left pane shows the metametamodel in terms of object and property types. MM, MO, MP, and MR denote resp. the MetaModel, MetaObject, MetaProperty, and MetaRole metaclasses. Every item is defined as an instance of itself, ensuring a reflexive definition. The middle pane depicts the definition of a simple statechart metamodel and the right pane shows how this metamodel can be used to instantiate a simple statechart (`idle`$\rightarrow$`busy`).

5 The Reconciliation Framework

We first formally describe the hypothesis of the world in which users *play*. We note U the set of all the possible users who can play some roles in a cooperative work. They can join or leave the game (i.e. cooperative work) whenever they want. Let's call R the databases/repositories users work with. The relation $use \subseteq U \times R$ denotes the use of a database by a user; given that databases are not shared between users. Each database is filled to store a MetaL specification (i.e. a set D). We note $data : R \rightarrow T \times L$ this process, where T is an infinite set of names and L denotes all the sets Ds (see Section 4) that fulfil the axioms

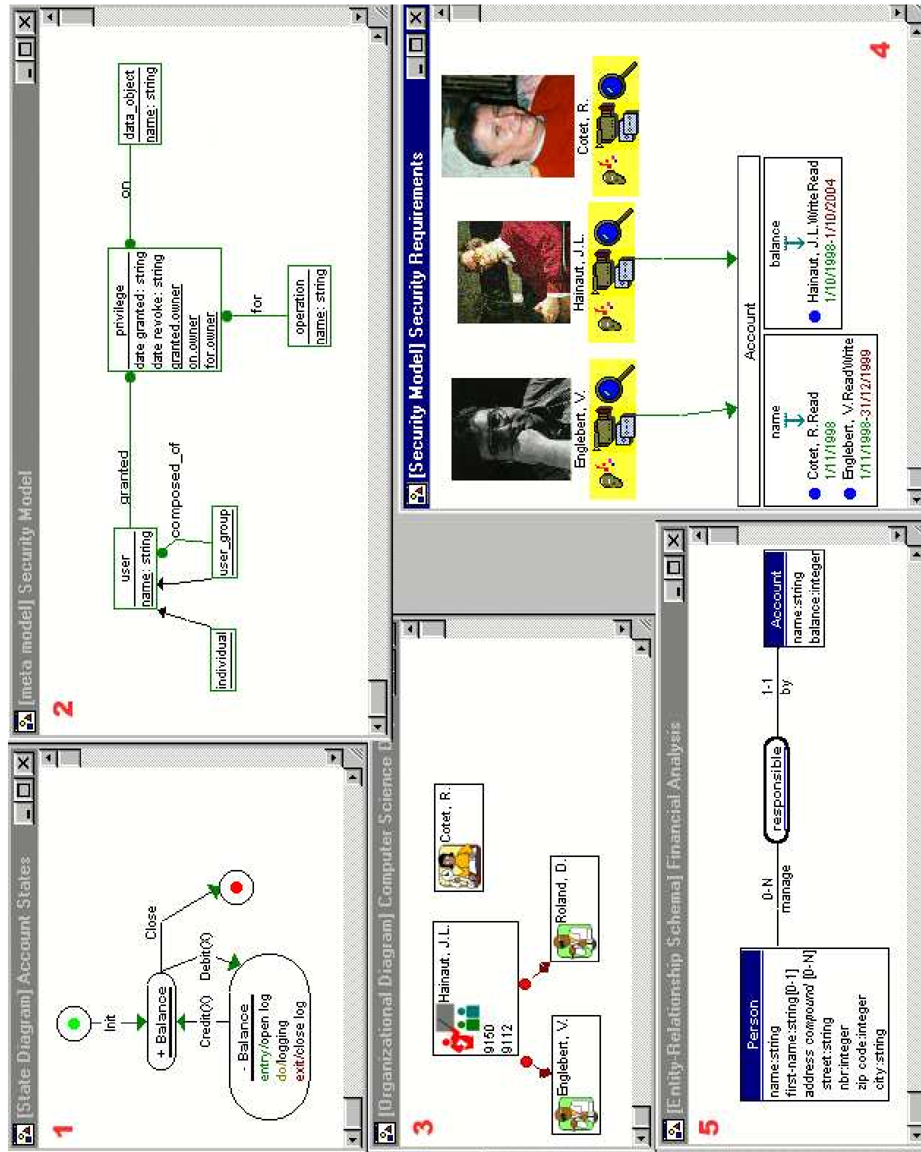

Fig. 5. DSM examples modelled with MetaL$_2$. Window 2 shows the metamodel of a DSL to specify security privileges related to database schemas. Model 4 attaches privileges on elements of model 5 to users of the organisational model 3. Model 1 is a classical UML like statechart.

of the MetaL language. At this time, the name in T denotes the repository from where data is coming from. The initial situation of our scenario would be described as $use = \{(A, R_A), (B, R_B)\}$ where Alice (A) and Bob (B) are resp. using repositories R_A and R_B. They are populated as: $data(R_A) = \{r_A\} \times D_A$, $data(R_B) = \{r_B\} \times D_B$ where $D_{A,B} \in L$.

The cooperation scenario is implemented by the exchange of a repository chunk from A to B. This chunk is a subset S of $data(R_A)$ that contains information relevant for the user and that meets the axioms of MetaL[2]: it is cohesive and autonomous. The chunk is built by a fixpoint process that improves the chunk at each step in order to meet the axioms. This process is explained hereafter:

> Let's first define some preliminary functions: function $\gamma : 2^D \to \mathbb{B}$ returns true if the information is consistent wrt. the MetaL axioms. Function $\delta : 2^D \to 2^D$ computes which elements from D are lacking in order to make the argument "more" consistent, indeed, the addition of new elements can in turn infringe other axioms. The following properties are satisfied: $P_1 \equiv \gamma(D) = \text{true}$, and $P_2 \equiv \forall X \subseteq D : \delta(X) \subseteq D$. If C: $C \subseteq D$ denotes the elements of interest to user A, then the chunk can be computed as:
>
> **while not** $\gamma(C)$ **do** $C := C \cup \delta(C)$
>
> This computes the smallest consistent set S containing C. The $\delta(X)$ function is built from rules such as: **if** $\exists x \in X \cap O,\ \exists ot \in OT \setminus X : iof(o, ot)$, and features of ot are used in X, **then** $\delta(X)$ must return ot. From properties P_1 and P_2, we see that *a)* such a set exists, *b)* that the process terminates, and *c)* that the result is contained in D.

When Bob receives S, the problem consists in adding S to $data(R_B)$. But this last process must make sure to preserve the distinct identity of the data that would be specific to A and to merge the common data between A and B — for instance, elementary types such as object types `String`, `metaclass`, etc. We suppose that a function $I_R : D \mapsto \mathbb{Z}$ maps every item to a unique negative number if this item is a common knowledge. Otherwise it maps every item to a unique positive number. Of course, there are as many functions I_R as there are repositories[3]. If we keep the information about the original repository with every data, we can now merge S and $data(R_B)$ as:

$$data(R_B) \cup \big\{(r, d) \in S \,|\, I_{R_A}(d) \geq 0\big\}$$

An action is an elementary access to the repository (create, read, update, delete) with additional information. Let's name *Action* the set of actions that can be performed on one element.

Bob can now modify his copy of the chunk: create/read/update/delete (CRUD) data at the model and metamodel levels. Once this job is finished, the modified

[2] i.e. $\{d | \exists(r, d) \in S\} \in L$.

[3] These functions can be defined as an increment, and then, several repositories could hold the same data while building distinct functions I.

chunk can be sent back to Alice. As in [20], we suppose that all the CRUD actions on the repository are logged and stored in a journal. All events related to an item are listed and the sequence of modifications can be reiterated. This enables the re-creation of models at different stages of their development. It also contains important milestones such as sending and receiving of method chunks or even free users annotations. When receiving the journal J_B from Bob, entries from the local journal and J_B can be compared to decide if J_B can be replayed while preserving the actions operated by Alice. This decision could be defined in order to preserve a strong serialization between the transactions [4], but weaker definitions could be considered depending on the granularity of the data to allow an interleaved execution for instance. The consistency rules should be checked at the end of this process. Since the concepts of instances and types are considered as first-class data, the merging of both journals according to the serialization rules concerns both models and metamodels at the same time.

Figure 6 presents how users Alice and Bob can work asynchronously on either M or MM. In this figure, labels noted "__" denote information not relevant for our explanation. The top panel of every repository is considered as a common knowledge, and is thus mapped to a consistent negative value by functions I_x. The reconciliation process merges these panels. When the first chunk arrives in Bob's repository, the items are added and any confusion is avoided with the adornment of the origin repository name (the T element is the label on the left top corner). When Bob sends back the chunk, a second reconciliation occurs. By comparing the journals, the T element helps the process to recover the elements that come from Bob, possibly modified, but that should be merged with items already present on the Alice's side. The figure shows one possible result of the reconciliation process. Others could exist depending on the strategy used, as discussed previously.

Another issue is the consistency of the repository that may occur when a part is replicated and maybe modifed by concurrent users. The journals can help to detect such problems and even to solve them sometimes. But this action unfortunately occur a-posteriori and this may be too late. A "weight-watchers" technique [2] can be used to detect such problems a-priori. This consists in marking each data with a weight (say 256). This weight would next be distributed (e.g. division by 2) between the users if they share data. When a reconciliation happens, the data recovers its weight if the sender abandons his/her ownership. The use of shared data (i.e. its weight$<$256) can now be detected, and computations that would have side effects outside the method chunk can be detected and executed at his/her own risks. Figure 7 demonstrates this principle where at some time t, Alice and Bob share the same chunk. Every data of this chunk is then weighted $\frac{256}{2}$, if Alice decides in meanwhile to apply a transformation from that chunk to a relational model, then the output of this transformation can be invalidated later if the reconciliation process affects the chunk.

Fig. 6. The Reconciliation Scenario. Modified items are represented with bold letters or a thick line. This scenario doesn't illustrate items deletion.

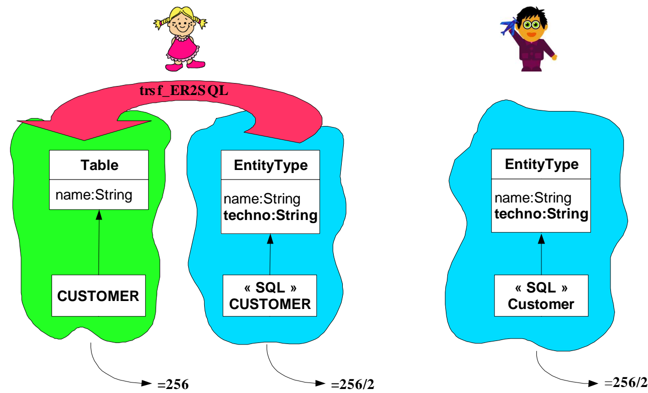

Fig. 7. Weight-Watcher Example. When Alice decides to transform the "CUSTOMER" entity type, the presence of partial weights in the argument of the transformation informs her that the argument could change once the transformation has been applied, impacting so the semantics of traceability between the models.

6 Results and Discussion

Our proposition allows the exchange of method chunks that consists in both models and metamodels data. The reconciliation of both models and metamodels may require to solve inconsistencies, redundancies and semantic mismatches. By preserving in a unique language the relationships between the instances and their types, we can provide a reconciliation process that encompasses several abstraction levels at the same time. To the best of our knowledge, [20] is the most complete work in that domain, but by generating explicitly a database schema from the metamodel, it breaks this homogeneity. Moreover, by avoiding the use of a central server (database or CVS-like), users can be autonomous and do not require any administration. In MetaEdit+ [1], import/export of chunks is possible as well as merging chunks together but there is a restriction: a user can not import back the chunk he has just exported. Nevertheless, this tool allows users to update the replica with patches, that scenario has not yet been considered in our framework.

We envision other benefits as the journals could be interactively reiterated to explain the contribution of the concurrent users or become first-class objects. The journal enables the recreation of models/metamodels at different stages of their development. Although our approach has been introduced with the MetaL language, it will be possible to extend this framework to other works such as RDF/OWL [22] or Telos [16].

7 Future work and conclusions

The support for cooperative tasks between DSM tools is still an open issue, especially for asynchronous collaboration. This cooperative approach can be achieved

by the exchange of method chunks. Our paper presents a theoretical framework that allows such exchanges at all the abstraction levels (models & metamodels) based on an ad-hoc meta data description language (MetaL).

This work is carried out in the context of the MetaDone DSM tool. It is based on MetaL and is currently being implemented in Java (30 KLOC). Although MetaL proposes interesting advantages over other DSLs like EMF or UML (reification, defining relationships between elements of distinct abstraction levels, management of all the abstraction levels in a homogenous way), these make the reconciliation process more complex. The proposed reconciliation framework is still at its early stage of design. Specific DSM reconciliation aspects should be considered: maintaining the consistency, refining the reconciliation oracle and explaining its strategies to the users, and defining a methodological framework for method chunks reconciliation.

References

1. *MetaEdit+ System Administration.* `http://www.metacase.com/support/45/manuals/sysadmin/sa-2_4_1.html`, 2009.
2. Maarten van Steen Andrew S. Tanenbaum. *Distributed Systems: Principles and Paradigms.* Prentice-Hall, 2002.
3. Christian Bartelt. Consistence preserving model merge in collaborative development processes. In *CVSM'08: Proceedings of the 2008 international workshop on Comparison and versioning of software models*, pages 13–18, New York, NY, USA, 2008. ACM.
4. Philip Bernstein. *Principles of Transaction Processing.* Morgan Kaufmann, San Diego, 1997.
5. H.T. Chou and W. Kim. A unifying framework for version control in a CAD environment. In *Twelfth International Conference on VeryLarge Data Bases*, pages 336–p344, Kyoto, 1986.
6. Juan de Lara and Hans Vangheluwe. Using atom^3 as a meta-case tool. In *ICEIS*, pages 642–649, 2002.
7. Eclipse. http://www.eclipse.org/.
8. Electronic Industries Association. CDIF Technical Committee. *CDIF Integrated Meta-model Common Subject Area*, eia/is-112 edition, December 1995.
9. Vincent Englebert and Patrick Heymans. Towards more extensible metaCASE tools. In A.L. Opdhal J. Krogstie and G. Sindre, editors, *International Conference on Advanced Information Systems Engineering (CAiSE'07)*, number 4495 in LNCS, pages 454–468, 2007.
10. Holt, Schürr, Sim, and Winter. GXL: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, April 2006.
11. Steven Kelly. Case tool support for co-operative work in information system design. In Colette Rolland, Yu Chen, and Meiqi Fang, editors, *Information Systems in the WWW Environment*, volume 115 of *IFIP Conference Proceedings*, pages 49–69. Chapman & Hall, 1998.
12. Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling. Enabling full code generation.* Wiley-IEEE Computer Society Pr, 2008.
13. Takashi Kiriyama, Tetsuo Tomiyama, and Hiroyuki Yoshikawa. A model integration framework for cooperative design. In Duvvuru Sriram, Robert Logcher, and

Shuichi Fukuda, editors, *MIT-JSME Workshop*, volume 492 of *Lecture Notes in Computer Science*, pages 126–139. Springer, 1989.
14. Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment.
15. Andrea De Lucia, Fausto Fasano, Giuseppe Scanniello, and Genny Tortora. Enhancing collaborative synchronous uml modelling with fine-grained versioning of software artefacts. *J. Vis. Lang. Comput.*, 18(5):492–503, 2007.
16. J. Mylopoulos. Conceptual modeling and telos. In P. Loucopoulos and R. Zicari, editors, *Conceptual Modeling, Databases, and CASE. An Integrated View of Information Systems Development*, chapter 2, pages 49–68. John Wiley & Sons, Ltd, 1992.
17. OMG. *MOF 2.0/XMI Mapping Specification, v2.1*, formal/05-09-01 edition, 2005.
18. PNML. http://www.pnml.org/.
19. Jolita Ralyté and Colette Rolland. An approach for method reengineering. In Hideko S. Kunii, Sushil Jajodia, and Arne Sølvberg, editors, *ER*, volume 2224 of *Lecture Notes in Computer Science*, pages 471–484. Springer, 2001.
20. Motoshi Saeki. Configuration management in a method engineering context. In Eric Dubois and Klaus Pohl, editors, *CAiSE*, volume 4001 of *Lecture Notes in Computer Science*, pages 384–398. Springer, 2006.
21. Kjeld Schmidt and Liam Bannon. Taking CSCW seriously: Supporting articulation work. *Computer Supported Cooperative Work*, 1:7–40, 1992.
22. W3C. The world wide web consortium (w3c) — http://www.w3c.org.
23. Jing Zhang. Metamodel-driven model interpreter evolution. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 214–215, New York, NY, USA, 2005. ACM.