Ontology-based Model Synchronisation

Federico Rieckhof, Mirko Seifert, and Uwe Aßmann

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
federico.rieckhof|mirko.seifert|uwe.assmann @tu-dresden.de

Abstract. Models are the central artifact in Model-Driven Software Development (MDSD). Being defined by meta models, they expose a strict syntactic structure. This property allows for processing models mechanically and using generic tools for arbitrary models. However, the well-known meta modelling languages (e.g., Essential MOF (EMOF)) do not have a formal semantical foundation. Rather, the semantics of models is defined implicitly and informal in respective specification documents. To resolve this restriction, the integration of models and ontologies has been proposed earlier [1]. In contrast to models, ontologies do have a formal grounding and allow for more sophisticated reasoning about the information represented therein.
This paper presents an approach to employ the ontological representation of models for synchronising changes across related models. Existing ideas from ontology mapping and alignment are applied to model synchronisation to investigate to what extent these technologies can be used in MDSD.

1 Introduction

With the advent of MDSD, code is replaced by models as the primary artifact in software development processes. By using standardised meta modelling languages (e.g., EMOF) the structure (i.e., the abstract syntax) of such models can be formally defined. If augmented with a description of the concrete syntax, either graphical or textual, models can be created and edited. To obtain actual software products from models, the latter are usually transformed over multiple stages. Starting from very abstract descriptions, more information is added yielding refined (i.e., more concrete) models. The most concrete models are then transformed to source code, which implements desired functionality.

When looking at models involved in such a development process one can detect lots of redundancy. For example, information is duplicated by the transformations between models. Even though one should strike for as little duplication as possible, there is often sound reasons for giving up this goal in favour of other objectives. For example, different views on the same data enable to focus on certain aspects of a system without worrying about others. This is the central idea behind using Domain-Specific Languages (DSLs) to model a system. Each of the DSL models shares some information with other models, yet it allows to model a specific concern (e.g., the user interface or the persistence strategy) separately. In other cases, information may be duplicated because tools can operate only on specific data formats and conversion and thereby duplication is needed to use these tools. A last reason for introducing redundancy in software artifacts is performance. Complex analysis of models (or source code) often requires to extract relevant information in order to perform the analysis in reasonable time.

Despite the sound reasons for introducing redundancy, there is the strong need to maintain consistency among all software artifacts. Inconsistent models must yield inconsistent implementations, which will definitely cause systems to fail. So whenever redundancy is introduced, action must be taken to make sure consistency preservation for this particular duplicated information is ensured.

In the MDSD community various approaches have been developed to solve this problem. Trace information is collected to explicitly capture the relations between duplicated information [2]. Bidirectional transformations (e.g., Relations in Query View Transformation (QVT) [3]) are used to synchronise models. Most prominently Triple Graph Grammars (TGGs) [4] were proposed to synchronise models by means of a declarative description of the relation.

Recent research has shown that the strength of modelling languages (i.e., the precise specification of abstract syntax) can be augmented with the power of formal semantics. In [5] the integration of ontologies and the Unified Modeling Language (UML) was shown, in [6] similar was achieved for DSLs. Opposed to pure models based on object-oriented meta modelling languages (e.g., EMOF), the semantics of the models is no longer informal or implicit, but rather formally defined and explicit. Ontologies and the Web Ontology Language (OWL) [7] being a wide-spread representative thereof, seem suitable for such an integration.

We believe that this shift toward explicit, formal model semantics can foster solutions to the model synchronisation problem. In particular we expect the following benefits from using ontologies in the context of model synchronisation:

1. Reduce specification effort - Existing synchronisation approaches need complex specifications. Ontologies may help here because implicit information can be derived by reasoners.
2. Intuitive specification - OWL-DL uses a very intuitive representation of knowledge. Classes are sets of individuals and subclasses are sub sets of their super classes. This may ease the understanding of models mapping.
3. Bidirectionality - Existing unidirectional synchronisation approaches require separate specification for each synchronisation direction. The declarative style of ontologies may support the specification of both directions at once.
4. Reuse ontology mapping technology - The whole existing body of work in the area of ontology mapping can be applied to obtain model mappings.

In this paper we investigate the first three points. To do so we introduce a running example in Sect. 2. Before presenting our OWL-based approach to model synchronisation in Sect. 4, we will recapitulate the mapping of models to ontologies in Sect. 3. This mapping has been discussed earlier [1, 8], but ours differs slightly, which is why we outline the differences and the rationale behind them. In Sect. 5 we discuss related work and conclude in Sect. 6.

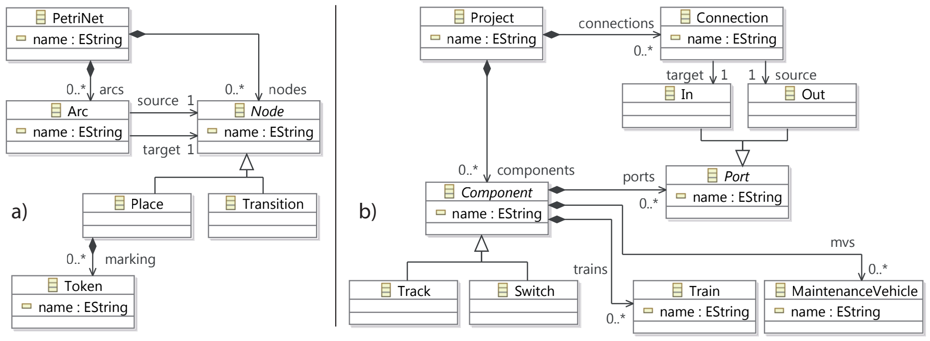

Fig. 1. Meta Model of Petri Nets (a) and Toy Train Models (b).

2 Running Example

To illustrate our approach we will use a basic scenario where two domain models need to be kept in sync. This scenario is taken from [9] and has been slightly extended by adding the meta class `MaintenanceVehicle`. The two involved domains are *petri nets* and *toy trains*. The goal of the synchronisation is to make sure the petri net model implements the dynamic semantics of the train model.

The meta model for petri nets is shown in Fig. 1a). Every petri net consists of arcs and nodes. Arcs do connect two nodes in a given direction. Nodes can be either places or transitions. The former can hold multiple tokens. A constraint not depicted in Fig. 1 forbids that arcs connect two nodes or two transitions. In addition, petri nets, arcs, nodes and tokens have a name.

The meta model for toy trains is depicted in Fig. 1b). Each toy train project consists of components and connections. Components can be either tracks or switches and expose ports. Connections connect ports, the direction is here given by making a distinction between ports of type `In` and `Out`. Tracks do have exactly one incoming and one outgoing port, whereas switches can have two ports of the same type (i.e., either two `In` and one `Out` port or the other way around). Furthermore, trains and maintenance vehicles can reside on tracks and switches.

The mapping between the two domains can be put as follows. Each petri net relates to one toy train project. Each track relates to an arc. `Out` ports relate to places and `In` ports to transitions. Connections between ports do also relate to arcs. A more complicated mapping is needed for switches. Switches with two incoming ports relate to an arc connecting a transition and a place. Switches with two outgoing ports relate to two arcs that connect a single transition with two places. An example of two synchronised models in shown in Fig. 2.

The figure uses concrete graphical syntax for both models. To make the mapping more obvious the components of the toy train model (shown in Fig. 1b) are also depicted in grey shade in Fig. 1a)). This way it is easier to see which parts of the petri net correspond to which parts of the train model.

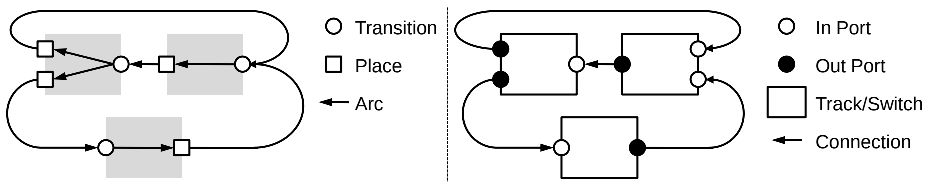

Fig. 2. Example of Synchronised Petri Net (left) and Toy Train Model (right).

3 Bridging Ontologies and Models

To use ontological tools for the synchronisation of models, a bridge is needed to transfer models to the world of ontologies. Ontology tools and services operate on ontology repositories and a process is needed to present models to these services. The mapping of models and meta models to OWL has been discussed in [1] and generalized to other types of ontological spaces in [8]. A mapping of UML to ontologies has been presented in [10], but this is not as relevant here as we are only interested in mapping the concepts of meta modelling languages (i.e., EMOF). Thus, this section mostly repeats parts of [1] and [8], which we found necessary to understand the mapping between models and OWL—a prerequisite for Sect. 4.

For our experiments we used Ecore on the modelling side—the Eclipse implementation of the EMOF standard. Therefore, we will restrict the mapping in this section to this particular meta modelling language. For other meta modelling languages a similar mapping might be suitable. On the ontology side, we used OWL-DL, which is why the mapping will refer to OWL concepts in the following. We will not map the concepts `EOperation`, `EFactory` and `EParameter`, because these are not relevant to model synchronisation. For the sake of simplicity we will also assume that domain meta models span one `EPackage` only.

As EMOF (and Ecore as its implementation) is an object-oriented modelling approach, the dominant concepts are classes and objects. Classes are also a concept in OWL, objects correspond to OWL individuals. Both Ecore and OWL do have the notion of `instance-of`, but with a different semantics. Ecore objects can be instance of at most one class unless classes are related by inheritance. In this case, objects are also instances of all super types of their class. In OWL individuals can be instances of multiple classes even if these are not related by inheritance. To correctly represent the semantics of meta models in OWL, classes are declared to be disjoint if there is no inheritance relation between them.

Attributes of Ecore classes (i.e., `EAttributes`) can be naturally mapped to OWL data properties. Opposed to attributes, data properties are not bound to one specific class in OWL, but can exists on their own. This can cause a name clash if two Ecore classes define attributes with the same name. When translating meta models to OWL qualified names can be used to solve this issue. References connect classes in Ecore and can be represented by OWL object properties. Again, name clashes can occur which the mapping tool must take care of.

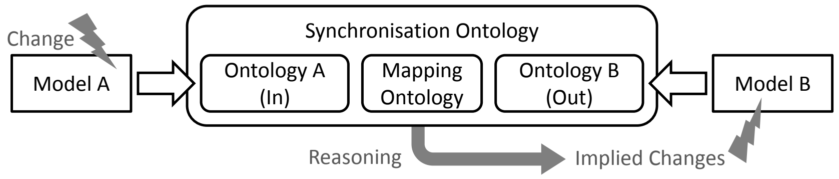

Fig. 3. Schematic View of the Synchronisation Procedure.

To return from OWL to the modelling world, a reverse mapping is needed. This is not possible in general. However, if the OWL knowledge base conforms to our construction schema, models can be created from OWL data. The constraints must be met, because they explicitly model the implicit semantics of models. If an OWL knowledge base does not do so, the corresponding model is not valid w.r.t. the semantics of models. Furthermore, only concepts that do have an equivalent in the modelling world can be used. Since we will use OWL behind the scenes only, this can be easily ensured.

Equipped with this mapping one can either implement a bridge based on (1) adaptors, build one that relies on (2) transformations or (3) fully integrate model and ontologies as shown in [6]. We choose to use a transformation-based bridge (2) because this was readily available from earlier work performed in the EU MOST project[1]. Nonetheless, the synchronisation mechanism presented in Sect. 4 is equally applicable if models are adapted to their ontological representation. The adaptors perform the transformation on-the-fly while we do execute it whenever we need to switch between the two technological spaces. We must emphasise that our focus is not building this bridge, but rather to investigate benefits that can be gained from using ontology tools for synchronising models.

4 Synchronising Models using Ontologies

Being equipped with the bidirectional mapping between models and ontologies, one can now focus on model synchronisation, which is essentially about propagating changes that are made to one model to all related models. In the scope of this paper, we will restrict the synchronisation to two models. So whenever a change is made to one of the two, appropriate changes need to be applied to the other one. As we want to use ontological tool machinery to do so, changes need to be transferred to the respective ontology, before the set of implied changes can be determined by reasoning. This procedure is depicted in Fig. 3.

After the models and the performed changes are transferred to the respective ontology a mapping between the two modelling domains is needed. This information is captured in a so-called *Mapping Ontology*, which is similar to transformation specifications in the modelling worlds. This ontology refers to the involved meta models by using the concepts that were derived from the meta

[1] http://www.most-project.eu

models. Note that this mapping is handcrafted as it requires distinct knowledge of the involved domains (i.e., the respective modelling languages).

In this section, we will first discuss possible mappings between meta models in terms of the derived ontologies in Sect. 4.1. For different types of mappings possible realisations are presented. Based on the domain ontologies (i.e., the ones derived from the meta models and the models) and the mapping ontology a reasoner can infer information that is needed in order to synchronise the models. Details of this procedure can be found in Sect. 4.2.

4.1 Mapping Specification between Models using Ontologies

OWL-DL—being the concrete ontology language used in this paper—is based on Description Logics and is therefore strongly connected to set theory. OWL classes basically denote sets of individuals. When we speak of synchronising ontologies (i.e., the ontologies derived from our models), we refer to the synchronisation of individuals. We are interested in the actions that must be taken if a new individual is created in one of the two domains. The relation between the sets (i.e., the classes) defined in each ontology is the key to this decision. We will therefore closely look at typical types of relations that do exist between domain classes and how they can be handled in terms of set definitions. Thus, the general idea behind the mapping specifications, which will be presented shortly, is based on the consideration of classes as sets of individuals. The objective of synchronisation mappings is therefore to define the relations between these sets.

When looking at relations between objects one can distinguish different types of mappings depending on the number of related individuals. For example, relations between individuals can be pairwise—one individual of one domain corresponds to exactly one in the other domain. An individual can also be related to a group of individuals in the opposite domain. In the most general case groups of individuals are related to each other. These three main relation types form the first column in Table 1. The first type (i.e., the 1:1 relation between individuals) can be further split into sub types depending on the number of classes involved in the mapping. These types are best explained using our running example.

Individual Relation	Class Relation	Example Relation (Individuals)
1 : 1	1 : 1	petriNet - project transition - inPort place - outPort
	1 : n	token - (train, maintenanceVehicle)
	n : m	N/A
1 : n	1 : n	connection - (arc, transition, place) switch - (arc, transition, place) track - (arc, transition, place)
n : m	n : m	N/A

Table 1. Types of Mappings between Domain Models.

Let us first consider the relation between petri nets and toy train projects. Each petri net corresponds to one project. If we denote the individuals that form a pair as *left* and *right*, all left individuals belong to the same class (from the left domain) and all right ones belong to another class (from the right domain). The same can be observed for the relation between transitions and `InPort`s, as well as places and `OutPort`s. The situation is slightly different for tokens, trains and maintenance vehicles. Here, individuals again form pairs, but classes differ within one domain. Some tokens correspond to trains, others to maintenance vehicles. The more general case where classes differ in both domains cannot be found in the running example. If petri nets would have different kinds of tokens, which would correspond to trains and maintenance vehicles we could observe the general case. This is not the same as having one type of token corresponding to trains and another one to maintenance vehicles.

Having seen the different types of mappings, the question is how to implement such a mapping in terms of OWL constructs. In this paper, we will present two approaches to specify mappings. The first one is based on defining mappings using subclass definitions. The second one employs Semantic Web Rule Language (SWRL) [11] rules to define model relations. Even though the latter approach subsumes the former w.r.t. expressiveness, we will present both, because we consider the subclass definitions to be very natural and more easy to understand.

Mappings based on Subclass Definitions OWL classes and properties can be considered as sets. Each set contains all the individuals that are instances of the class or property. When synchronising models using OWL, a mapping is needed, which allows to determine all the individuals that are required in the opposite domain to restore a synchronous state. To determine these individuals the formalisation of the relation between classes is needed. For the most basic mapping case (cf. Row 1 in Table 1), classes can be defined as being equivalent. In this case, subclasses are not even required.

The second row of Table 1, contains the first type of mapping which can be handled by defining subclasses. To specify that some tokens are mapped to trains and others are mapped to maintenance vehicles, we introduce two subclasses of `Train`. One that denotes the set of tokens that corresponds to trains, another one that represents all tokens that are related to maintenance vehicles. To elaborate this in more detail consider Fig. 4.

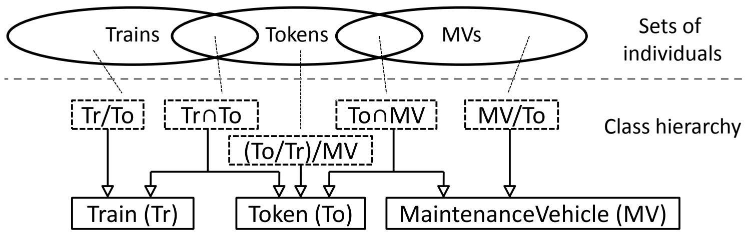

Fig. 4. Mapping Subsets of Classes.

The domain classes (`Train`, `Token` and `MaintenanceVehicle`) are depicted as boxes with solid border. The additionally defined subclasses use dashed borders. In Fig. 4 five such subclasses are depicted. The first one (`Tr/To`) denotes all trains that do not have a corresponding token. In our example this set is empty, but for other mappings it might be not. The second subclass (`Tr∩To`) represents all trains that correspond to a token. The definition of this class is crucial for the example mapping. The concrete definition determines which tokens map to trains and which do not. The third subclass (`(To/Tr)/MV`) contains all tokens that are neither mapped to trains nor to maintenance vehicles. Again in our example this set is empty, but this may not be the case for other mappings. The two remaining classes (`To∩MV` and `MV/To`) are reverse to the first two.

By defining subclasses that extend classes from both domains, the pairs of individuals are represented as one individual belonging to two classes. For example, a train-token pair is represented by an individual of class `Tr∩To`. If a train individual is created, one can reason that it also belongs to class `Tr∩To`, which in turn implies that it is an instance of class `Token`. We will use this in Sect. 4.2.

The mapping based on subclass definitions does work for the first three types shown in Table 1 (i.e., all cases where pairs of individuals are related). It cannot be applied if multiple individuals take part in a relation, because domain classes are defined to be disjoint within one domain. Thus, we can only define non-empty subclasses that extend classes from different domains. Extending two classes from the same domain yields an empty set of individuals, because the super classes are disjoint. Besides this restriction, the approach seems very intuitive. Once the designer of a mapping has decided that individuals are related pairwise, she must clarify which set of individuals is mapped to which class in the opposite domain. The resulting set definitions can then be formalised as OWL class definitions. We believe that splitting classes into subsets, where each subset corresponds to one class in the opposite domain, is easy to understand. Besides its simplicity, another advantage of the subclass mapping approach is its bidirectionality. The mapping is independent of the synchronisation direction. We can add individuals to any domain and infer missing ones in the opposite domain.

Mappings based on SWRL rules A second approach to express mappings is based on SWRL. SWRL is an extension of OWL that enables the use of Horn-like rules in combination with the OWL knowledge base. In this paper, we will restrict ourselves to the DL-safe subset of SWRL. A SWRL rule can be read as follows. If all atoms in the antecedent are true, then the consequent must also be true. Using SWRL rules one can specify dependencies between domains. For example, whenever a petri net contains an arc that connects a transition to a place, a corresponding track must exist, expressed by the following rule:

$$\begin{gathered} Arc(?x) \wedge Place(?y) \wedge Transition(?z) \wedge \\ petriIn(?x, ?y) \wedge petriOut(?x, ?z) \rightarrow \\ Track(?x) \wedge InPort(?y) \wedge OutPort(?z) \wedge \\ havePort(?x, ?y) \wedge havePort(?x, ?z) \end{gathered} \tag{1}$$

One can read rule 1 as follows. All individuals of type `Arc`, which are related with one `Place` and one `Transition` by the object properties `petriIn` and `petriOut`, imply that the individuals of type `Arc` are also of type `Track`, the individuals of type `Transition` are also of type `OutPort` and that the individuals of type `Place` are also of type `InPort`. Furthermore, the object property `havePort` must be true. As this rule does only define the dependency in one direction, a second rule is needed to express that each track and its respective ports must be mapped to an arc connecting a transition and a place in the petri net. The rule to express this is as follows:

$$\begin{gathered} Track(?x) \wedge InPort(?y) \wedge OutPort(?z) \wedge \\ havePort(?x, ?y) \wedge havePort(?x, ?z) \rightarrow \\ Arc(?x) \wedge Place(?y) \wedge Transition(?z) \wedge \\ petriIn(?x, ?y) \wedge petriOut(?x, ?z) \end{gathered} \tag{2}$$

As one can see, both a forward and a backward rule is needed to synchronise individuals of type `Track` from the toy train meta model. Using rule 1 and rule 2, a reasoner is able to infer the synchronisation in both directions without any problems. More simple relations between classes can also be expressed. For example, the next two rules state that `Place`s and `OutPort`s are equivalent.

$$Place(?x) \rightarrow OutPort(?x) \tag{3}$$
$$OutPort(?x) \rightarrow Place(?x) \tag{4}$$

This case can be handled more easily using subclass definitions as this does not require separate specifications for each direction. But, to clarify the relation between both approaches, it must be said that the use of SWRL rules subsumes the subclass definition approach in terms of expressiveness.

Returning to the first two rules (i.e., the ones mapping arcs to tracks), one may wonder whether these two rules interfere with other rules (e.g., the ones for mapping arcs to switches). Similar to tracks, switches with two incoming ports are mapped to an arc that connects a transition to a place. But, not only switches are mapped to arcs. Connections between toy train components are also mapped to arcs. So, how can one distinguish between arcs that represent tracks, arcs that correspond to switches and arcs that are related to connections?

Let us have a look at the initial rules for mapping arcs and connections:

$$Arc(?x) \rightarrow Connection(?x) \tag{5}$$
$$Connection(?x) \rightarrow Arc(?x) \tag{6}$$

Suppose there is an arc that connects a transition and a place. Obviously both rules (rule 1 and rule 5) can match. Thus, a reasoner will infer that the arc has both type `Track` and type `Connection`. If there is also a rule for mapping switches, it might in addition infer that the arc is also of type `Switch`. As our domain classes are disjoint by definition, this is obviously a contradiction.

To resolve this problem, we propose to combine the subclass approach and the SWRL rules. Instead of using the classes defined by our meta models in the SWRL rules, we use distinct subclasses. So instead of mapping all arcs to connections (as in rule 5) and all connections to arcs (as in rule 6), we map only subsets, i.e., the arcs that represent connections and, respectively the connections that represent arcs. This yields the following rules, which replace rule 5 and 6:

$$Arc2Connection(?x) \rightarrow Connection2Arc(?x) \tag{7}$$

$$Connection2Arc(?x) \rightarrow Arc2Connection(?x) \tag{8}$$

The class `Arc2Connection` denotes the arcs that map to connections, class `Connection2Arc` is the set of connections that map to arcs. The same procedure is applied to all other rules concerned with mapping the same set of classes. By introducing subclasses we resolve the contradiction faced before. However, to make this work, the relation between the newly introduced subclasses and the original classes (derived from the meta model) must be defined. For example, all connections are mapped to arcs. Therefore, class `Connection2Arc` is equivalent to class `Connection`. This is not the case for arcs. Here we must specify, which arcs are mapped to connections and which represent tracks or switches.

To do so, we define subclass restrictions. For example, class `Arc2Connection` is defined as a subclass of `Arc`, but with the restriction that is must connect a place to a transition. All other types of arcs (i.e., the ones mapped to tracks and switches) do connect transitions to places, which is the other way around. To distinguish arcs that must map to tracks and switches, other conditions are specified. For arcs representing tracks the source transition must have exactly one incoming and one outgoing arc. For the two types of switches, the situation is different. Here, the source transition can have two incoming arcs (for switches with two `in` ports) or two outgoing arcs (for switches with two `out` ports). The restrictions can be implemented using OWL or SWRL—whatever is more easy.

Once the subclasses are restricted like this, the reasoner can infer which subclass a concrete arc belongs to and consequently which rule applies. By the combination of SWRL rules and the subclass construction we can synchronise individuals which do have different counterparts depending on their context. This allows to handle both the 1:n and n:m mapping cases from Table 1.

Handling Primitive Data Types In addition to classes and relations, meta models can define attributes, which have a primitive type. These attributes can be mapped to data properties, when translating models to ontologies. When synchronising models, one must take care of these attributes and their values. In principle, there are two cases that can appear when handling primitive types.

In the first case attributes are mapped directly. For example, the name attribute of a class in one meta model is mapped to the identifier attribute of another meta class. In this case no processing is applied to the attribute value. To handle such mappings the previously described approach for synchronising classes and object properties can be employed. Based on the definition of sub properties the sets of attribute values can be distributed across the classes that need to hold these values. For example, the name of class `Token` can be split into two sub properties (i.e., the names for trains and for maintenance vehicles).

The second case can be observed whenever values need to be processed (e.g., set conditionally or concatenated). In this case we define sub data properties and process these using SWRL built-ins. Because operations performed by built-ins can do arbitrarily complex computations, there is no way to avoid explicit specifications for each transformation direction. For example, if strings are concatenated by a rule, another rule is required to specify how strings are split.

To give an example lets suppose we would like to prefix the names of all arcs that correspond to a track with the string `track_`. To do so, rule 9 and rule 10 can be added. Here, rule 9 concatenates the prefix and the value of data property `name`. The result of the concatenation is set as value for property `subArcName`.

$$\begin{gathered} Track2Arc(?t) \wedge name(?t, ?name) \wedge \\ stringConcat(?newName, "track\_"\hat{}\hat{}string, ?name) \rightarrow \\ subArcName(?t, ?newName) \wedge Arc2Track(?t) \end{gathered} \tag{9}$$

In addition to rule 9, another rule is needed that implements the opposite mapping. Here, the prefix is removed from the name of the arc. The remainder is set as value of data property `subname`, which is the sub property of name in the train ontology.

To summarise, one can say that the handling of primitive types is not much different from traditional model synchronisation approaches. Primitive types are often processed using complex and often non-invertible operations. This does leave no other choice than specifying individual rules for each transformation direction. Nonetheless, we believe that support for bidirectional built-ins in SWRL would be beneficial in this context. Such built-ins must then be evaluated similar to relations (i.e., compute values for unbound parameters).

4.2 Propagating Model Changes

In the previous section two different possibilities to specify mappings between meta models using ontological concepts have been presented. Based on subclass definitions and SWRL rules one can now specify desired mappings between domain models. Once these mappings have been formalised, the question is how changes that are made to models can be processed by the ontological tool machinery. Thus, this section will give details about this procedure.

The first logical step required, when synchronising changes across models is to capture the changes made. This step is rather a technical issue and its concrete implementation depends on the editors used to modify models. In Fig. 5 this first step is depicted on the left. Once information about changes is available, we employ an OWL reasoner[2] to infer implicit information. More concretely, we let

[2] our prototypical implementation uses Pellet [12]

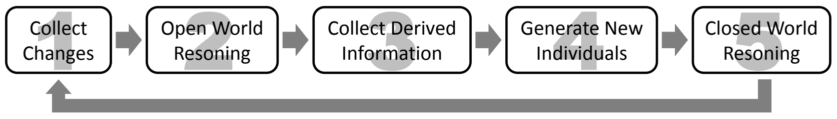

Fig. 5. Change Handling Procedure - Overview.

Pellet perform the *realisation* task. The goal of this task is to calculate all types for all individuals. To do so, the reasoner must also perform a *classification*, which calculates all subclass and equivalence relations between OWL classes. While performing these tasks, Pellet uses the rules defined in the specification mapping to imply new facts or to derive subclass and equivalence relations. Thus, additional types besides the ones explicitly defined by the model to OWL transformation are assigned to individuals. When looking at the mapping types presented in Sect. 4.1, one can see that these were intentionally designed to allow a reasoner to infer the information needed to synchronise the domain models.

The result of performing realisation and classification using Pellet is a large graph. This is the outcome of step 2. To extract information that is of interest to us, step 3 analyses this graph and extracts a tree structure. To do so, first, all individuals and all concrete classes are collected by sending a query to Pellet. Using these two sets, we iterate over all individuals and determine their concrete classes using a SPARQL[3] query. This data is the first level of our tree structure.

Next, all secondary classes (i.e., the classes we have introduced in addition to the original classes derived from the domain meta models) are collected. To do so, again the set of all classes for each individual is determined. From this set all original classes are removed to obtain secondary classes only. These sets form the second level of the tree structure. Finally, the object and data properties for all individuals are collected and put into the third and fourth layer of the tree.

By traversing this tree structure from top to bottom we can derive the set of individuals that need to be created. These new individuals are then explicitly added to the knowledge base and the respective models. So, after step 4 a new, incrementally updated, synchronous version of the domain ontologies is obtained. For all new individuals a trace is kept. Thus, if an element is deleted in the source model, the corresponding target model elements can be removed as well.

To ensure the consistency of domain models, the constraints are checked by closed world reasoning using Pellet ICV[4]. By using closed world reasoning, some issues can be detected that are ignored by the checks performed before. For example, lower bounds for multiplicities are checked in this step. If a transition does have dangling reference, this is not reported by the previous checks, but will issue an error when using Pellet ICV. If no errors are detected, new individuals can be transferred to the respective models yielding a synchronous state again. For subsequent changes the procedure starts from the beginning.

[3] SPARQL Protocol and RDF Query Language [13]
[4] http://clarkparsia.com/pellet/icv

5 Related Work

Model Synchronisation and Transformation Model synchronisation can be performed in various ways. Often ad-hoc approaches based on imperative languages are used. These are hard to analyse and usually restricted to the unidirectional synchronisation. Therefore declarative approaches should be favoured. Here, one can either use pairs of unidirectional model transformation rules (e.g., written in Atlas Transformation Language (ATL) [14] or QVTO [3]) or write bidirectional rules (e.g., using QVTR [3] or TGGs [4]). The latter choice has the appealing property that both directions are defined by a single specification. This does of course ease maintaining synchronisation definitions.

Compared to OWL, being based on Description Logics, most model transformation approaches expose a less formal grounding. ATL does not have a formal semantics at all. QVTR defines its semantics by a mapping to QVT Core, but there are still open questions and contradictions in the standard [15]. The approach with the best formal grounding are TGGs, as they are based on graph rewriting. The approach presented in this paper and TGGs are similar in terms of expressiveness. Our running example was actually taken from a paper on TGGs. However, the point of using semantical techniques for models synchronisation, is not to gain expressiveness, but to ease specification. In this regard, our approach differs from TGGs, as it uses a different specification mechanism based on subclasses. Whether this is more intuitive than writing TGG rules lastly depends on the user, but at least an alternative way to do so was made available.

Ontology Matching and Alignment The ontology community has put a lot of effort into methods to match ontologies. The goal of that work is to generate a mapping that formalise a common subset of concepts or terms. Different approaches based on string comparison, language analysis, structural matching, machine learning and combinations thereof [16], have been evaluated in this area. While all these matching algorithms and systems can be employed to synchronise models once models were transformed to ontologies, their use is orthogonal to the work presented in this paper. Our objective is to show how a handcrafted mapping specification can help to synchronise models, in particular how a reasoner can be used to derive the information needed to keep models in sync. Our focus is therefore on synchronising the individuals rather than matching the classes in the ontologies to each other. Nonetheless, the ontology matching approaches mentioned before can assist the creation of such a mapping, but this is an optional process that can be performed before, if feasible.

To represent mappings we used OWL and SWRL, which does have some drawbacks (see [16] Sect. 8.1). Using other mapping formalisms like C-OWL [17] is also possible. For the scope of this paper, we decided to stick with languages, where reasoner technology was readily available to obtain a practical prototype.

6 Conclusion and Future Work

In this paper we presented an approach to synchronise models using their ontological representations. Based on existing work, models were mapped to ontologies and vice versa. To specify the synchronisation of models within the technological space of OWL, we presented several types of mappings that can be employed. Depending on the type of relation needed, either subclasses definitions can be used to map model elements or, for more complex structural relations, SWRL rules can formalise relations between domain models. The former idea is based on the notion of sets formed by individuals of classes. Subclasses are defined to divide sets and equivalence relations map subsets to corresponding sets of individuals in the opposite domain. We defined types of mappings that can be handled with this approach and other classes that cannot. The latter can be handled using SWRL rules, which sacrifices the bidirectionality that was available when using subclass definitions. We have implemented the running example based on OWL and the Pellet reasoner to justify the approach practically.

In Sect. 1 our objectives were defined as *reducing specification effort*, *increasing intuitivity of specification* and *obtaining bidirectionality*. Our results do confirm these partially. We found the mappings based on subclass definitions very intuitive, but unfortunately these can only handle pairwise mappings between individuals. Also, the need for bidirectional specifications was met by the subclass approach. But, more complex mappings using SWRL rules do not posses this property. Whether specification effort is reduced, cannot be determined at this point. The technical obstacles one needs to deal with (e.g., when defining OWL classes instead of directly working on the meta models) are still to large to gain benefit from the formal semantical foundation. Modelling languages with built-in support for semantics [18] seem essential to overcome this barrier.

To further investigate the use of ontological tool machinery for model synchronisation, we plan to apply the mappings to an industrial case study, where synchronisation has been realised before using model-based synchronisation tools. The comparison of both approaches will then gain more insight about both ways to ensure model consistency. In addition the tight integration of meta models and formal semantics as proposed in [5, 6, 18] will be used to perform synchronisation directly on model instances. This will hopefully eliminate the need of transforming models to OWL. We believe that adding formal semantics to the modelling world is more rational than using ontologies to represent system models, simply because of the great amount of mature and powerful tools that are readily available to create and process models.

In this paper we restricted ourselves to mappings that are explicitly defined by domain experts. While this is feasible for small mappings, a lot of effort is needed for meta models of industrial size. Therefore, using ontology matching techniques to derive initial mappings, which can then be refined by domain experts, is another interesting objective for future research.

Acknowledgement

This research has been co-funded by the European Commission within the FP6 project Modelplex #034081 and the FP7 project MOST #216691.

References

1. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: MoDELS. Volume 4199 of LNCS., Springer (2006) 528–542
2. Drivalos, N., Paige, R.F., Fernandes, K.J., Kolovos, D.S.: Towards Rigorously Defined Model-to-Model Traceability. In: Proceedings of the 4th ECMDA Traceability Workshop (ECMDA-TW 2008), 9-12 Jun 2008, Berlin, Germany. (2008)
3. The Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. Technical report (April 2008)
4. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Proceedings of 20th International Workshop on Graph-Theoretic Concepts in Computer Science, Herrsching, Germany. Volume 903 of LNCS., Springer Verlag (1994)
5. Parreiras, F.S., Staab, S., Winter, A.: TwoUse: Integrating UML Models and OWL Ontologies. Technical Report 16/2007, Institut für Informatik, Universität Koblenz-Landau (2007)
6. Walter, T., Ebert, J.: Combining DSLs and Ontologies Using Metamodel Integration. In Taha, W.M., ed.: Domain-Specific Languages. Volume 5658 of LNCS., Springer (2009) 148–169
7. W3C: OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. Technical report (October 2009)
8. Parreiras, F.S., Staab, S., Winter, A.: On Marrying Ontological and Metamodeling Technical Spaces. In Crnkovic, I., Bertolino, A., eds.: ESEC/SIGSOFT FSE, ACM (2007) 439–448
9. Kindler, E., Wagner, R.: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical Report tr-ri-07-284, University of Paderborn, D-33098 Paderborn, Germany (June 2007)
10. Gasevic, D., Djuric, D., Devedzic, V.: MDA-based Automatic OWL Ontology Development. Software Tools for Technology Transfer **9**(2) (2007) 103–117
11. W3C: SWRL: A Semantic Web Rule Language Combining OWL and RuleML - W3C Member Submission. Technical report (May 2004)
12. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web **5**(2) (2007) 51 – 53
13. W3C: SPARQL Query Language for RDF. Technical report (January 2008)
14. ATLAS Group: ATLAS Transformation Language (ATL) User Guide. http://wiki.eclipse.org/ATL/User_Guide (February 2006)
15. Stevens, P.: Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. In Engels, G., Opdyke, B., Schmidt, D.C., Weil, F., eds.: MoDELS. Volume 4735 of LNCS., Springer (2007) 1–15
16. Euzenat, J., Shvaiko, P.: Ontology matching. Springer, Heidelberg (2007)
17. Bouquet, P., Giunchiglia, F., van Harmelen, F., Serafini, L., Stuckenschmidt, H.: C-OWL: Contextualizing Ontologies. In Fensel, D., Sycara, K.P., Mylopoulos, J., eds.: International Semantic Web Conference. Volume 2870 of LNCS., Springer (2003) 164–179
18. Walter, T., Parreiras, F.S., Staab, S.: OntoDSL: An Ontology-Based Framework for Domain-Specific Languages. In Schürr, A., Selic, B., eds.: MoDELS. Volume 5795 of LNCS., Springer (2009) 408–422