Representation and Traversal of Large Clabject Models

Thomas Aschauer, Gerd Dauenhauer, Wolfgang Pree

C. Doppler Laboratory Embedded Software Systems, University of Salzburg
Jakob-Haringer-Str. 2, 5020 Salzburg, Austria
firstname.lastname@cs.uni-salzburg.at

Abstract. Multi-level modeling using so-called clabjects has been proposed as an alternative to UML for modeling domains that feature more than one classification level. In real-world applications, however, this modeling formalism has not yet become popular, because it is a challenge to efficiently represent large models, and providing fast access to all information spread across the meta-levels at the same time. In this paper we present the model representation concept that relies on a permanent condensed view of the model, the corresponding traversal algorithms, and their implementations that proved adequate for model-driven engineering of industrial automation systems consisting of hundreds of thousands of model elements.

Keywords: Clabject, Multi-Level Modeling, Efficient Representation

1 Introduction

For the development of software intensive systems, model-driven engineering (MDE) is a promising approach for handling their inherent complexity. For real-world applications, MDE requires adequate means for describing the system's essential properties. In particular for domains that feature more than one classification-level, also known as meta-level, prominent modeling languages such as UML [1] fall short and workarounds are required [2]. Multi-level modeling, as an alternative to UML, is able to handle multiple domain meta-levels within a uniform framework [3]. Advantages of such a modeling approach have been shown by several contributions [2, 4, 5].

In real-world applications, however, multi-level modeling has been barely applied. Major hurdles for adopting a multi-level formalism are the lack of (1) available modeling environments that allow rapid prototyping, (2) real-world applications that corroborate the benefits of multi-level modeling, and (3) efficient implementations that are capable of handling large models. This paper focuses on (3) and briefly touches (2). Examples of (1) are described e.g. by Gutheil et al. [4].

We have applied multi-level modeling for automation systems in the domain of combustion engine development. There we use MDE to generate configuration parameters for the automation system. As it turned out in practice, the classification hierarchy supported by multi-level modeling is crucial for building and maintaining concise models. For the model transformations and for end-user views, however, it is often necessary to have a "condensed" view such that for a certain model element all

structural properties are easily accessible, rather than having to traverse the whole meta-level hierarchy to collect that information. What makes matters even more complicated is that this condensed view in practice is not only used for read access, but is also modified. This implies that a method is needed for transparently mapping modification operations on the condensed view to the classification hierarchy.

This paper shows how to efficiently store and traverse a multi-level model, which is also capable of handling modification operations. Together with an efficient representation, we are able to provide a permanent condensed view of the model, which turned out to be the preferred access method for end-users. Since our models typically are large, that is, in the order of hundreds of thousands of model elements, we validate our performance goals by using sufficiently large test models.

In the following we present the basics of multi-level models in our domain and key requirements for their representation and retrieval. We then describe our representation method and a traversal algorithm. Finally, we evaluate our implementation.

2 Multi-Level Modeling with Clabjects

Automation systems in our domain are inherently complex for various reasons. They are usually built individually and comprise a large number of ready made parts, which are often customized. They also integrate sophisticated measurement devices that are software intensive systems by themselves. In this section we briefly describe the multi-level modeling approach that was employed to cope with that complexity [5].

Multi-level modeling is an alternative approach to conventional modeling that is able to overcome the limited support for modeling domain metalevels. The basic idea of multi-level modeling is to explicitly represent the different abstraction levels of model elements. Assume, for example, that we have to model concrete combustion engines, but also families of combustion engines from different vendors that specify the properties of the individual engines, in UML. Conceptually, engine is an instantiation of its family. Since instantiation is not directly supported at the M1 layer [1], workarounds such as the type-object pattern are required [2].

Different flavors of multi-level modeling have been proposed as solutions. Atkinson and Kühne, for example, propose a uniform notion of *classes* and *objects*, known as a *clabject* [2], that allows for an arbitrary number of classification levels; its advantages are well documented [3, 2, 4]. In principle, a clabject is a modeling entity that has a so-called *type facet* as well as an *instance facet*. It thus can be an instance of a clabject from a higher level, and at the same time it can be the type for another clabject instance at a lower level. Figure 1 shows the clabject model of our combustion engine example. The notation used here is similar to that of the original clabject concept, that is, a combination of UML notations for classes and objects [2, 6]. Each model element has a compartment for the name, and a combined compartment for the type facet and the instance facet. The arrows between the levels represent the "instance of" relationship. At the domain metatype level, the types Engine, Diesel Engine and Otto Engine are modeled like a conventional class hierarchy. Their *fields*, which are the equivalent of attributes in multi-level modeling [2], such as Inertia and Preheat_Time, are part of the corresponding clabject's type facet.

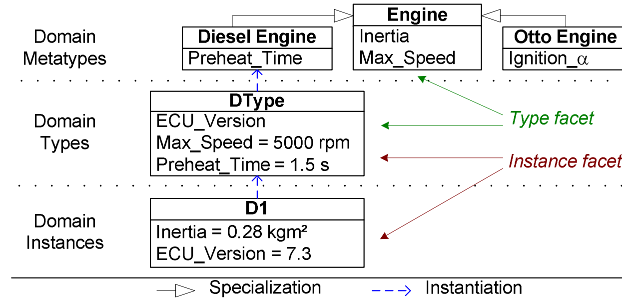

Fig. 1. Clabject-based engine model

Specified at the domain type level, the clabject DType is an instance of Diesel Engine. It provides values for the fields Max_Speed and Preheat_Time, which are part of DType's instance facet, and introduced a new field ECU_Version, which is part of DType's type facet. The domain instance D1 in turn instantiates DType and provides values for Inertia and ECU_Version. Note that D1's type facet is empty. By definition, the clabjects at the top-level only have a type facet, whereas the clabjects at the bottom level only have an instance facet.

3 Model Representation and Traversal Requirements

Analyzing the intended uses of our models, we can identify a number of requirements and assumptions regarding the usage of the models and expected performance and space requirements. This guides the design and implementation of the actual internal representation as well as the traversal algorithms.

(I) Large models. Due to the inherent complexity of the domain, we expect the models to be large, that is, consisting of more than 100,000 elements. This implies that we have to be able to represent models of considerable size in a memory-efficient way.

(II) Structural similarity. Automation systems in our domain typically are usually built individually of ready made parts, which are often customized. A substantial amount of these ready made parts, however, have similar structural information and share the same field values. For example, most temperature sensors are of the same type and thus the internal structure of their models is equivalent, except for some customization such as an additional plug. As an example, consider figure 2.

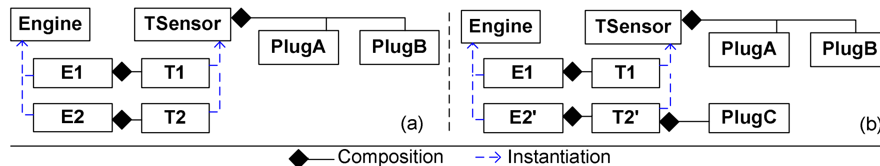

Fig. 2. Multiple usage of sensor type: (a) unmodified, (b) with modification

Here the sensor type TSensor is instantiated multiple times. In case (a) the sensor type TSensor is used within the context of both, engine E1 as clabject T1, and engine E2 as clabject T2. In case (b) the sensor type TSensor is instantiated twice, but with a modification: in T2' the element PlugC is added. The rest of the information modeled in TSensor, i.e. the containment of PlugA and PlugB, is the same for both instances.

(III) Model traversal. The prospective users of models are either the end users, building or exploring the model in a graphical user interface, or the model transformation system, analyzing the model and applying transformations. In both cases, the main method for accessing model elements is through traversal, starting at the root of the containment hierarchy, and visiting connected and contained model elements. In contrast to random access, one does not access contained model elements directly. So in our example in figure 2 (b) PlugC is not accessed directly, but only by navigating from E2' to T2' and then to PlugC.

We can distinguish two different ways of traversing a model: First, we can follow the *connectors*, which are the equivalent of associations in multi-level modeling [2]; for our example this corresponds to navigating from E2' via T2' to PlugC. Second, we can follow the instantiation and the generalization relationships; for the same example, this corresponds to navigating from E2' to Engine or from T2' to TSensor. Both traversals reveal essential information. Since for some uses, such as the model transformation, the complete model has to be traversed, it is crucial that the traversal of large models can be done in a reasonable amount of time.

(IV) Condensed traversal. The model transformation, for example, focuses on the structure of a particular model element. It does not matter whether the structure is modeled at a certain model element itself, or whether is received via inheritance or instantiation. In other words, this requires a traversal by following the connectors, but also by incorporating the connectors that are instantiated or inherited. For users performing this kind of "mixed" traversal, i.e. following connectors and also the instantiation and inheritance relationships, it is necessary to transparently "flatten" the classification hierarchy during traversal to provide a *condensed view* on the model.

As an example, consider the model shown in figure 2 (a). When the model transformation performs a condensed traversal, the information that both engines E1 and E2 use the same definition of TSensor is not relevant. What is relevant is the fact that both engines have a sensor with two plugs. So for the model transformation, all structural information modeled in TSensor appears as if it was modeled directly in E1 and E2. In the end, the result of the traversal looks like each engine defines its own sensor, as shown in figure 3 (a) and (b).

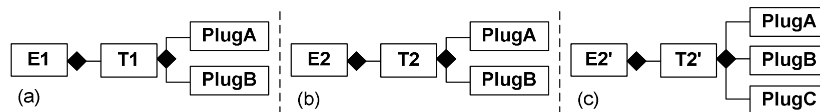

Fig. 3. Transparent traversal result: (a) starting at E1, (b) starting at E2, and (c) starting at E2'

When we traverse the model starting at E1 we get the associated clabject T1, and in turn the clabjects associated with T1, i.e. PlugA and PlugB. Starting the traversal at

E2 yields an analogous result. Note that although in the model PlugA only appears within the definition of TSensor, in the overall traversal result the same plug appears twice: within E1, and within E2. In both occurences, however, PlugA represents different real-world elements and thus actually has different identities, stemming from the semantics of the composition aggregation. The information about the classification, however, is not lost but available on request for each model element. For the elements T1 and T2, for example, it is possible to retrieve their types, i.e. TSensor. For both appearances of PlugA the retrieved type is the PlugA contained in TSensor.

For the case when the instantiation of TSensor is accompanied by a modification, as presented in figure 2 (b), we get the traversal result as shown in figure 3 (a) and (c). Again, starting the traversal at E1 yields the same result as described above. Starting the traversal at E2', however, yields a different result: First we get the associated clabject T2', and in turn the associated clabjects PlugA, PlugB, and PlugC. Note that PlugC appears in the traversal result in the same way as PlugA and PlugB do.

The algorithms necessary for condensed traversals could, of course, be implemented by the model transformation itself. It turns out, however, that this kind of traversal is also required by our user interface, so the modeling environment supports condensed traversal as the default.

(V) Modifiable traversal result. When a user traverses the model, the traversal result has to be modifiable, independent of the kind of traversal performed. While this is straightforward for the traversals following either connectors, or instantiation and inheritance, it is more difficult for the condensed traversal method. Assume, for example, that a user traverses the model shown in figure 2 (a), which leads to the condensed traversal result show in figure 3 (a) and (b). Further assume that the user adds a plug named PlugC to T2. Performing this operation on the traversal result implies that the modeling environment has to store the difference between the original element, which is TSensor, and its modified usage, which is T2. The expected effect on the traversal result is that the plug is retrieved additionally to the plugs already defined in the original sensor, which is exactly what we have already seen in figure 3 (c). Technically this requires determining the involved classification level and adding the modified elements there, such that we get the model as shown in figure 2 (b).

4 Implementation

The goals for moderate memory consumption (I) and good traversal performance (III) are contradicting, as keeping hundreds of thousands of individual elements in memory does not scale. So we have to trade memory for traversal speed. It turns out that the structural similarity in real-models (II) is a property that can be used to save memory, since we have to store structural information only once and reference that information when similar structures are modeled. For traversing the model in order to get the condensed view (IV), however, the saved memory implies some performance penalty since we have to reconstruct the structure of a clabject from the instantiation and inheritance relationships. Since the elements retrieved by the traversal have to be

modifiable (V), we must add certain information such that the link between the classification hierarchy and the condensed view does not get lost, as shown in the sequel.

4.1 Language Representation

The modeling environment has to provide the language models are built of. As such it must be capable of representing arbitrary models, model elements, their fields, and relationships between them. Furthermore, multiple classification levels have to be supported, so means for expressing instantiation and generalization have to be provided. The basic entities of our modeling language are Clabject, Connector, Field, and Data Type; they are shown in figure 4.

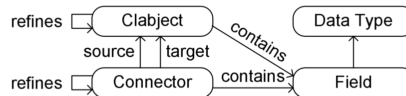

Fig. 4. Representation of language elements

The elements Clabject, Field and Connector are typical for clabject-based modeling languages [7]. What is unique by our solution is that the representation uses one single *refines*-relationship to represent both, instantiation as well as inheritance, for clabjects and connectors. We denote an element that is the source of a refinement relationship, i.e. a type or a generalization, as *refined clabject*, and the target of a refinement relationship, i.e. an instance or a specialization, as *refining clabject*. By using this single relationship we do not claim that the semantics of instantiation and inheritance are similar [8]. For the sole purpose of representing large models in memory and traversing them, however, a uniform treatment is beneficial, as we will see.

4.2 Permanent Condensed View

It is important to note that the language as described above is the interface of the modeling environment, i.e. model elements are represented as clabjects, connectors, and fields. Due to the traversal requirements as outlined earlier, we know that condensed traversal is the primary access method. Thus in our implementation condensed traversal is not just another way of exploring the model, but it is built in as foundation of the modeling environment. Its realization, however, requires some optimized data structures to be able to provide the condensed view within reasonable bounds of runtime and memory consumption. This, however, can be hidden from the users of the modeling environment.

4.3 Traversal of Refinements without Modifications

The simplest case of refinement (remember, this is either instantiation or inheritance) is refining an element without adding any additional information, neither additional

structural information, nor any values for fields. An example of non-modifying refinement is shown in figure 2 (a).

Let C be the set of all clabjects and $x_0 \in C$ be a refining clabject that refines $x_1 \in C$, which in turn refines $x_2 \in C$, etc., such that we get the sequence of refined elements (x_1, x_2, …, x_n), where $x_n \in C$ is an element that is not refined from any other element. Since neither inheritance nor instantiation allows circularity, n denotes the depth of the refinement path and is a finite integer with $n \geq 0$. Further let R be the mapping of a clabject to the sequence of refined elements, e.g. $R(x_0) = (x_1, x_2, \ldots, x_n)$.

The basic scheme for the traversal is to visit the clabject at the root of the containment hierarchy, all its contained clabjects, and subsequently all clabjects in the refinement path. For compositions of refined clabjects we have to take special care since the contained elements can appear several times in the traversal result. An example is the double occurrence of PlugA in figure 3 (a) and (b). The identity of these elements is not only defined by their refined element, but also by the "context" in which they appear. Since the refinement depth can be greater than one, the context is given by the sequence of refining elements. In figure 3 (a) the context for PlugA is given by the sequence *(T1)*; in figure 3 (b) the context is given by the sequence *(T2)*.

Since such clabjects actually do not exist, but appear only in the traversal result, we call them "virtual" clabjects. Virtual clabjects are temporarily represented by lightweight placeholder objects that are created on the fly during traversal. The garbage collector can dispose of them after the traversal has finished. We get the following algorithm for determining the condensed traversal:

traverseClabject(*x, ctx*) performs a condensed traversal for the clabject *x* with context *ctx*, which is a list of clabjects, by visiting the clabject itself and then subsequently all clabjects along the refinement hierarchy. For a non-refining clabject the context *ctx* is the empty list. The *add…*–calls denote that a node in the traversal is reached and that it should be added to the traversal result. The symbol ↻ denotes recursion.

1. Visit the clabject; note that for virtual clabjects the context defines its identity:
   a) *addClabject(x, ctx).*
2. Visit fields, including that of refined clabjects:
   a) ∀ *field a* ∈ *getFields(x):*
   b) *addField (a).*
3. Visit contained clabjects, including that of refined clabjects:
   a) ∀ *(connector r, context c)* ∈ *getCompositions(x, ctx):*
   b) *addConnector(r).*
   c) ↻ *traverseClabject(r.target, c).*

getFields(*x*) collects all fields of clabject *x* by following the refinement path. The result is a list of fields. The symbol || denotes concatenation of lists.

1. Iterate over the refinement path, including x, to find fields of refined clabjects.
   a) ∀ *clabject q* ∈ *x* || *R(x) :*
   b) *set result* ← *result || q. Fields.*

getCompositions(*x, ctx*) collects all compositions of clabject *x* with context *ctx* along the refinement path. Returns a list of pairs of the composition and the context.

1. Iterate over the refinement path, including x, to find compositions.
   a) $\forall$ *clabject* $q \in x \,||\, R(x)$:
   b) $\forall$ *connector* r, $r.source = q \wedge r.kind = Composition$
   Add the composition to the result; remember that the context of a refined clabject is the sequence of the refining clabjects in the current refinement path:
   c) *set result* $\leftarrow$ *result* $||$ $(r, cxt \,||\, R(x).firstUntil(q))$.
   (The list returned *by firstUntil(q)* does not include q, and is empty if $q \notin R(x)$.)

4.4 Modification and Materialization

The second case of refinement occurs when a refining element adds further information. The example of figure 2 (b) shows that the refining element T2' adds a plug to the structure of the refined clabject TSensor. Representing this case is straightforward since all the information about the modification is located at the refining element. More complicated is the situation where the information about the modification is not located in the refining element. Consider the example of figure 5.

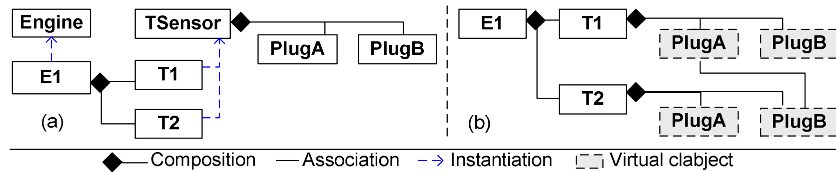

Fig. 5. Engine with two sensors: (a) model and (b) condensed view

The left hand side (a) shows the model of an engine E1 that contains two instances of TSensor, namely T1 and T2. The right hand side (b) shows the condensed view containing the virtual clabjects for both sensors. Now assume that we want to connect PlugA of T1 with PlugB of T2. Since neither of these two plugs exists as a clabject, i.e. both are virtual clabjects in the condensed view, we have to transparently create some sort of *proxy* elements that can be endpoints of connectors. We call this process "materialization", and figure 6 shows how it works.

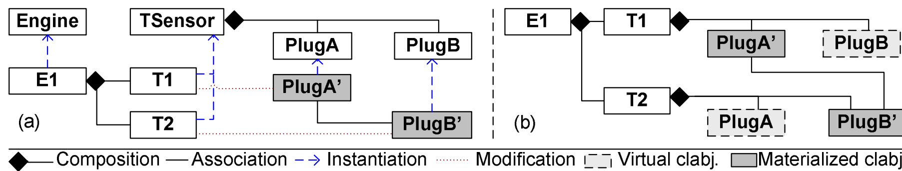

Fig. 6. Materialization: (a) model and (b) condensed view

In order to be able to connect the two plugs, we first have to create the materialized representation of the corresponding virtual clabjects, as shown on the left hand side (a): PlugA of T1 is materialized as PlugA', and PlugB of T2 is materialized as PlugB'. Now PlugA' and PlugB' are instances of the corresponding plugs of TSensor. Thus during materialization we create refinements. These refinements, however, until

now do not contain any information except their identity, so they do not use much memory. After materialization, we can create a connector between PlugA' and PlugB'. In the condensed view, as shown on the right hand side (b), we then also get the condensed view of both materialized clabjects.

A materialized clabject, similar to a virtual clabject, only needs to keep track of the refined clabject and the context, which is a list of references to other clabjects, to be uniquely identifiable. In addition, it only stores the difference information to the refined clabject, so a materialized clabject also is a lightweight element.

4.5 Traversal of Refinements with Modifications

Traversing a model with simple modifications, i.e. additional contained elements directly at a refining clabject, is similar to traversing a model without modifications, but we also have to follow the additional compositions. It is easy to see that *traverse-Clabject* can already handle this case.

For refinements with materialized clabjects, as shown in figure 6, a materialized clabject is reached indirectly by traversing its refined clabject since we follow only compositions. Consider the traversal order E1, T1, TSensor, and PlugA. Our algorithm fails here since we expect to have PlugA' in the traversal result, and not PlugA. To resolve that situation, we have to follow the refinement relationship in the reverse direction, since then we can transparently skip PlugA in the traversal and instead visit PlugA'. To prevent an exhaustive search for determining the inverse of the refinement relationship, we use a dictionary *rmap* that maps the refined elements to the refining elements. This map includes the elements that are depicted by the "Modification"-relationship in the figure. In our implementation each clabject stores its own *rmap*, so we get the following traversal algorithm:

traverseMaterializedClabject(*x, ctx*) performs a condensed traversal for the clabject *x* with context *ctx*; can handle clabjects as well as materialized clabjects.

1. Visit the clabject; note that for virtual clabjects the context defines its identity:
   a) addClabject(x, ctx).
2. Visit fields, including that of refined clabjects:
   a) ∀ field a ∈ getFields(x):
   b) addField (a).
3. Visit contained clabjects, including that of refined clabjects:
   a) ∀ (connector r, clabject y, context c) ∈ getMaterializedCompositions(x, ctx):
   b) addConnector(r).
   c) ↻ traverseMaterializedClabject(y, c).

getMaterializedCompositions(*x, ctx*) collects all compositions of clabject *x* with context *ctx*. Returns a list of triples with: composition, target clabject, and context.

1. Iterate over the refinement path, including x, to find compositions:
   a) ∀ clabject q ∈ x || R(x):
   b) rmaps ← rmaps || q.rmap. (*q.rmap* retrieves the *rmap* of clabject *q*)
   c) ∀ connector r, r.source = q ∧ r.kind = Composition:

   When there is a materialized clabject for the target, take that instead of the target:

d) *if* $\exists$ *m, m* $\in$ *rmaps: m.contains(r.target) then*
e) *set result* $\leftarrow$ *result || (r, m.get(r.target), cxt || R(x).firstUntil(q)).*
f) *else*
g) *set result* $\leftarrow$ *result || (r, r.target, cxt || R(x).firstUntil(q)).*

Traversing Non-Composition Connectors. Until now we have only considered compositions for traversal. For implementing the function *getConnectors(x, ctx)* that retrieves all connectors for a given clabject, we have to distinguish several cases. Consider figure 7.

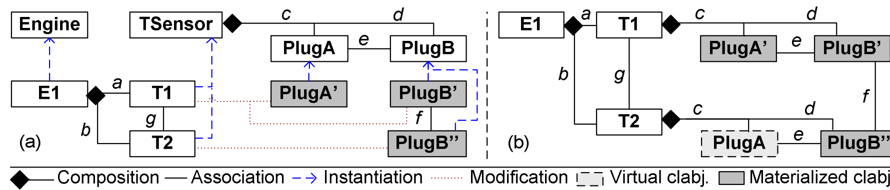

Fig. 7. Connectors and materialization: (a) model and (b) condensed view

Following the connector *g* in analogous to following compositions such *a* and *b*. For following direct connectors between materialized clabjects, such as *f*, we also have all information that we need. For following indirect connectors, i.e. connectors that stem from refined elements, between materialized clabjects such as *e* in the context of T1, additional information is required. Assume that during traversal we are at PlugA'. Following the refinement leads to PlugA, in turn to *e*, and further to PlugB. Now, however, we cannot resolve to PlugB'. Having the *rmap* of T1 enables us to resolve that materialization, and in the general case we have to use the whole context for resolution. Thus the context becomes an essential part of every materialized clabject. A similar case is following an indirect connector from a virtual clabject to a materialized clabject, such as *e* in the context to T2. Here we also need to resolve the materialized clabject by inspecting the context. Since virtual clabjects cannot have direct connectors (they must be materialized first), we have no further cases to consider.

Connector Refinement. Besides refining clabjects, also connectors can be refined. Handling these refinements requires resolving the refined connectors when we visit a connector during a traversal step. Analogously to refining clabjects, this only requires that we store the reverse refinement information at the corresponding clabject's *rmap*.

Field Refinement. Refining clabjects is not done as an end in itself, but typically is used to either provide field values in case of an instantiation, or to add new fields in case of a specialization. Thus the refinement relationship between two clabjects also relates their fields. Consider, for example, figure 7. Let TSensor declare the field Range. The instantiation of TSensor as T1 demands that we provide a range, e.g. from 0 to 100. Thus we can say that the field Range of T1 refines the field Range of TSensor. Extending *getFields* of our traversal algorithm is similar to extending *getCompositions* to *getMaterializedCompositions*.

Refinement Path. The presentation of the algorithms above used the function *R*, mapping a clabject to the sequence of refined elements. In our actual implementation we do not maintain a global map, but rather at each clabject and materialized clabject we store a reference to the refined clabject only. While this is beneficial for the performance, it is also necessary for the self-contained storage of the model parts, e.g. for building libraries of model elements. This, however, is out of scope for this paper.

4.6 Modifying the Condensed View

Since the condensed view is permanently available in our environment, we have to ensure that for any element that is visited via traversal, modification is possible. Modification for clabjects and materialized clabjects is straightforward, since these are exactly the places where we store the modification information. For modification of virtual clabjects, we first have to materialize them. Since for each virtual clabject the context is known, we already have all information that is needed to create the materialized clabject. So with our representation we have ensured that modifying the condensed view is possible, and moreover, that only local information is required.

5 Performance Evaluation

In order to demonstrate the feasibility of the internal representation and the traversal algorithm presented in the previous section, we performed measurements on test data.[1] We decided to use perfect n-ary trees in our tests. Informally, a perfect n-ary tree is a tree where all leaf nodes are at the same depth and each inner node, i.e. non-leaf node, has exactly n children. This decision to use such trees is based on the fact that we wanted to have (a) test structures of varying size, with (b) varying refinement depth. Furthermore, (c) the implementation of the condensed view does not depend on associations between model elements. In addition, we (d) want to use the same kind of test data for evaluating future extensions of the language. In our particular case, we used m trees, where each was a perfect quaternary tree of depth d. This test data construction simulates the existence of m top-level nodes in the model. The choice for quaternary trees is backed by the informal analysis of several models created with an earlier prototypical version of the modeling environment. Our tests use $m = 3$ and d ranging from five to nine, resulting in 4,095 to 1,048,575 clabjects.

Tests are performed for two principal cases: (a) trees without clabject refinement, and (b) trees where clabjects are refined to reuse structural information. In case (a), all clabjects are individually created. Thus in the traversal we do not encounter any virtual clabjects. In case (b), the structure of the lowest one or two levels is reused. If one level is reused, a clabject containing n leaves is created upfront. This is then reused for all clabjects at level $d - 1$. These clabjects thus contain n virtual clabjects

[1] All measurements were performed on a Dell Precision M65 Mobile Workstation, equipped with an Intel® Core™2 CPU operating at 2GHz and with 2 GB of main memory, and running Windows XP Professional. The runtime environment was Microsoft .NET 3.0. Tests were executed as individual processes to prevent side effects of e.g. the garbage collector.

each. In a subsequent test, we materialize them prior to traversal. Figure 8 shows the structure for both cases.

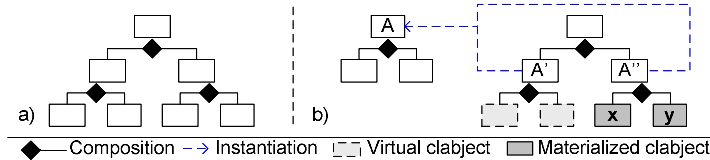

Fig. 8. Test data used for the measurements

The example on the left hand side represents case (a): a binary tree consisting of individually created elements. The example on the right hand side represents case (b): clabject *A* is created upfront, and is refined multiple times in the tree, resulting in virtual cjabjects as e.g. contained in *A'*. These virtual clabjects can be materialized, e.g. simply by renaming them to "x" and "y".

If two levels are reused, a similar element is created upfront, but containing *n* clabjects which in turn refine from a clabject containing *n* leaves. While in principle it is possible to create even deeper reuse-hierarchies, we restricted the reuse depth in our test cases since we think this is a realistic measure for real world models.

5.1 Traversal Performance

As outlined in section 3, the model transformation code is one of the use cases for our models. In order to verify the feasibility of our modeling approach with respect to the performance requirements, we performed traversal tests of the whole model. Traversal is done recursively on the condensed view, depth first, without performing any additional code besides book-keeping, such as counting the number of visited clabjects. The time is measured using .Net's built in *System.Stopwatch* class. Case (a), i.e. a model without reuse, is taken as baseline. Traversal times for case (b), i.e. the models with element reuse, are expected to be slightly slower, since according to the implementation described in section 4, the traversal has to create virtual leaf elements on the fly for each reused element. Traversal also has to keep track of the context information. Table 1 shows the corresponding measurement results.

Table 1. Model traversal performance

		Iteration Time [s]				
		No Clabject Refinement	Refining Clabjects at			
			1 Level		2 Levels	
Tree Depth	Number of Clabjects		Virtual	Materialized	Virtual	Materialized
5	4,095	0.02	0.02	0.02	0.03	0.02
6	16,393	0.05	0.06	0.06	0.08	0.07
7	65,535	0.14	0.24	0.24	0.31	0.29
8	262,143	0.53	0.92	0.93	1.19	1.12
9	1,048,575	2.15	3.53	3.65	4.74	4.54

Somewhat unexpected, traversing the tree with virtual clabjects is roughly equally fast as traversing the tree with materialized clabjects. We interpret the result in that it shows that the dynamic creation of virtual clabject on the fly is fast, while maintaining the context for virtual and materialized clabjects is an overhead. The time used for visiting one individual clabject does not increase with growing model size. More important, the numbers also show that our reuse approach is reasonable fast, both for interactive use at modeling time, where only small parts are traversed, and for use by the model transformation.

5.2 Memory Consumption

Besides traversal performance, memory consumption of the models is one of our main requirements. Another series of tests was thus performed, measuring the impact of creating models on the total memory used by the process. Memory consumption is measured by using .Net's built in *System.GarbageCollector* class before and after creating the model. The measured numbers thus represent the net size of the models. Case (a) again is the baseline. Models of case (b) are expected to consume significantly less memory than in case (a), since according to the implementation described in section 4, virtual clabjects require no memory except for the context information, and even materialized clabjects need to represent only incremental information. Table 2 shows the corresponding measurement results.

Models for case (a), i.e. without reuse, require significant amounts of memory. In our example, up to 868.28 MB are necessary even for a model containing only simple clabjects with basic fields such as a name. In contrast, models for case (b), which reuse clabjects, require significantly less memory. For elements with one reuse-level, all leaves of the tree are virtual clabjects and do not require memory except for the context information. As expected, e.g. the tree of depth 9 with virtual clabjects as leaves, requires about the same amount of memory as the similar tree of depth 8, consisting of individually created clabject only. Analogously, the tree of depth 9 with two levels of reuse requires about the same amount of memory as the similar tree of depth 7 without reuse.

Table 2. Model memory consumption

		Memory Consumption [MB]				
		No Clabject Refinement	Refining Clabjects at			
			1 Level		2 Levels	
Tree Depth	Number of Clabjects		Virtual	Materialized	Virtual	Materialized
5	4,095	3.37	0.85	1.90	0.22	1.67
6	16,393	13.23	3.42	7.31	0.85	6.36
7	65,535	53.54	13.42	29.52	3.42	25.78
8	262,143	215.01	54.30	119.26	13.43	103.23
9	1,048,575	868.28	218.02	478.94	54.30	415.23

The measurement was also performed where *all* virtual clabjects were materialized and thus exist as objects in memory. The memory consumption is still significantly

lower than for the models containing individually created clabjects. In real world models, however, we expect only a fraction of the virtual clabjects to be materialized, so this additional memory consumption is expected to be negligible. While memory consumption for models without reuse *is* problematic, we can see that our reuse approach keeps the memory consumption within practicable bounds.

6 Related Work

Handling of large models is a common requirement for the application of modeling environments in practice. The definition of "large", however, actually depends on the kind of models and on the subject domain. A natural border case is a model that barely fits into main memory. For the Eclipse Modeling Framework [9], for example, this problem also arises and is solved by dynamically loading and unloading model parts, transparently performed by the persistency layer [10]. EMF or MOF-based modeling approaches [11], however, do not support multi-level modeling with clabjects. In our implementation, we could exploit the property of structural similarity, which allows incorporating the space-efficient representation right at the implementation of modeling elements, so we can represent sufficiently large models without reaching memory limits.

The idea of unifying classes and objects has a long tradition in object-oriented programming languages, namely in prototype-based languages such as SELF [12]. A SELF-object consists of named slots that can carry values, which in turn are references to other objects. SELF uses an "inherits from"-relationship that unifies instantiation and specialization. Chambers et al. report on a similar assumption as we do: "Few SELF objects have totally unique format and behavior", since most objects are slightly modified clones [12]. They use so-called "maps" for representing common slots, such that individually created objects only have to store difference information. The basic idea is quite similar to ours, the implementation of a programming language, however, certainly differs from that of a modeling environment.

An early report by Batory and Kim on the quite complex domain of VLSI CAD applications also explores the structural similarity of model elements [13]. They describe an implementation based on a relational database that employs copying of data to achieve good retrieval performance. Their system, however, only supports one single classification level.

Gutheil et al. describe an effort to build a multi-level modeling tool that is also based on the clabject-idea [4]. They give some fundamental principles for coping with connectors in such an environment, e.g. for their graphical representation. It is however not reported how industry-sized models are handled.

7 Conclusion

This paper describes how core features of a clabject-based modeling environment can be implemented in practice. We describe the traversal algorithm for a condensed model view and how to reduce memory consumption of a condensed view. Based on

the theoretical part, we evaluated our approach with test models of varying size. The results show that our concepts and their implementations are efficient both with respect to traversal time and memory consumption. The resulting clabject-based modeling environment meets the requirements for a real-world application, and thus demonstrates that multi-level modeling can indeed be used for large industrial applications.

Acknowledgements
The authors thank Stefan Preuer for his excellent implementation of the clabject representation code and the traversal algorithms.

References

1. Management Group: Unified Modeling Language Infrastructure, v 2.1.2, (2007)
2. Atkinson, C. and Kühne, T.: Reducing accidental complexity in domain models. Software and Systems Modeling, Vol. 7, No. 3, pp. 345–359, Springer-Verlag (2007)
3. Atkinson, C. and Kühne, T.: The Essence of Multilevel Metamodeling. In proceedings of UML 2001, LNCS Vol. 2185, pp. 19–33 (2001)
4. Gutheil, M., Kennel, B, and Atkinson, C.: A Systematic Approach to Connectors in a Multi-level Modeling Environment. In proceedings of MoDELS'08, LNCS Vol. 5301, pp. 843–857 (2008)
5. Aschauer, T., Dauenhauer, G., Pree, W.: Multi-Level Modeling for Industrial Automation Systems. 35th Euromicro SEAA Conference, to appear (2009)
6. Object Management Group: Unified Modeling Language Superstructure, v 2.1.2 (2007)
7. Atkinson, C., Gutheil, M., Kennel, B.: A Flexible Infrastructure for Multi-Level Language Engineering, to appear (2009)
8. Kühne, T.: Contrasting Classification with Generalisation. In Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modelling, APCCM 2009, New Zealand, 2009.
9. Eclipse Foundation, Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf/
10. Stepper, E.: Scale, Share and Store your Models with CDO 2.0. Talk at eclipseCON (2009)
11. Object Management Group, Meta Object Facility (MOF) 2.0 Core Specification (2004)
12. Chambers, C., Ungar, D., and Lee, E.: An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. Lisp Symb. Comput. Vol. 4, No. 3, pp. 243–281 (1991)
13. Batory, D. S., and Kim, W.: Modeling concepts for VLSI CAD objects, ACM Transactions on Database Systems, Vol. 10, No. 3, pp. 322–346 (1985)