Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 253 (2010) 121–134
www.elsevier.com/locate/entcs

Library Concepts for Model Reuse

Markus Herrmannsdörfer[1] Benjamin Hummel[2]

Fakultät für Informatik
Technische Universität München
Garching bei München, Germany

Abstract

Reuse and the composition of libraries of partial system descriptions is a fundamental and well-understood practice in software engineering, as long as we talk about source code. For models and modeling languages, the concepts of reuse often are limited to *copy & paste*, especially when it comes to domain-specific modeling languages (DSLs). This paper attempts to give an overview of techniques for including support for reuse and library concepts both in the meta-model and the modeling tool, and presents a novel generative approach for this task. The technical consequences for each of the approaches presented are discussed and compared to each other.

Keywords: reuse, model library, model-based development

1 Introduction

A major trend in software engineering is the increased use of semi-formal models during software development. This includes models for capturing requirements, describing the software's design and architecture, or capturing a more formal specification of a system. These models are then used for different analysis and generation tasks which are not possible using documents written in natural language – such as checking consistency of different views of a system, generating source code or test cases from a model, or formal verification. While UML [8] has been a key player for driving model-based development in software development, in many domains the usage of so-called domain-specific modeling languages is more appropriate. These range from simple languages for the user interface design of digital watches [6], to more complicated ones for defining business processes [9] or developing software for embedded systems, such as the well-known example of Matlab/Simulink [11].

With these models getting more and more important and growing in size, reuse of common sub models becomes an important issue. Reusing well-tested parts of

[1] Email: herrmama@in.tum.de
[2] Email: hummelb@in.tum.de

doi:10.1016/j.entcs.2010.08.036

a model reduces the risk of introducing bugs and reduces the overall development costs. In the long term identifying commonly used parts and putting them into a library for later use – possibly adding a mechanism for parametrization – has lots of advantages over the commonly found reuse by just *copy & paste*. First, it makes reuse more structured and systematic, as it makes explicit which parts are actually designed to be used elsewhere – compared to just looting existing models. Second, it allows to better organize analysis and testing activities, by creating libraries of well understood and tested elements. Finally, it hugely simplifies dealing with change, both due to shifted requirements or detected bugs. Without a library, all duplicates of the model part being changed have to be found beforehand, which despite existing approaches for automation [2] is a tedious and error-prone [4] task. Thus we argue that meta-models and editors for domain-specific modeling languages should support structured reuse and library concepts.

Problem Statement

Although the need for reuse at the model level is often confirmed, only little work is available on the patterns and concepts used to integrate reuse and library support into a modeling language. Especially the impact which different realizations might have on the underlying meta-model and the tools manipulating and analyzing the models is rarely discussed.

Contribution

This paper attempts to give an overview of the possible choices for implementing reuse and library support into a DSL's meta-model and tool chain. Therefore we summarize a "well-known" technique for this task as well as one suggested in the literature, and introduce a (to the best of our knowledge) novel generative approach for achieving it. Our focus is especially on the discussion of the impacts and consequences implied by choosing one of them. The ideas and insights are mostly rooted in the current development of AutoFOCUS 3[3], a reimplementation of the AutoFOCUS research prototype for modeling embedded systems [10], and a proprietary editor for the COLA language [7] which was developed within the context of an industry project from the automotive domain.

Outline

The next section introduces our running example, which is used to demonstrate the impact of adding support for reuse. Sec. 3 gives an overview of existing approaches, while Sec. 4 introduces our solution to this problem. In Sec. 5 we discuss the consequences implied by each of them, before we conclude in Sec. 6.

[3] Actually, the concepts discussed here are part of the underlying framework called CCTS. Details are available at http://af3.in.tum.de/.

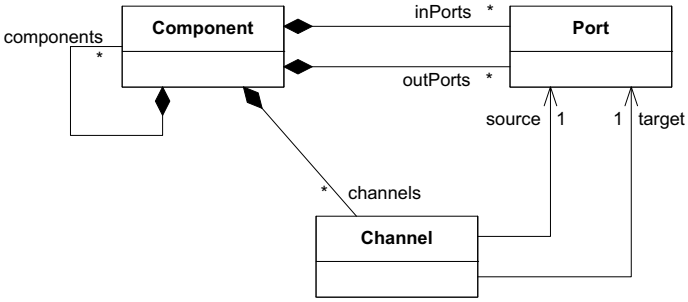

Fig. 1. Running Example Meta-model.

2 Running Example

We use a data-flow language for the component-oriented specification of systems as a running example throughout the paper. An excerpt of the language's meta-model is depicted in Fig. 1 as a UML class diagram. A Component defines a syntactic interface which consists of input and output Ports (compositions inPorts and outPorts). A component is either a basic component or composed of sub components, thus resulting in a component hierarchy. The sub components of a component are connected to each other via channels. A Channel of a component connects a source to a target port – with the restriction that only intra-level connections are allowed.

Reuse of components is a key technique to reduce the effort for the description of a system. Now we want to extend our modeling language in a way that it supports the reuse of components. In the following, components that can be reused are called component *types*. We should be able to make component types available through a so-called *library*. When it comes to reuse of a component, the reused component is called an *instance* of the component type. An instance of a component type can be used wherever a regular component can be used, *e.g.*, in the composition of another component. An instance has to be aware of its component type, so that it can adapt itself to changes of this component.

We deliberately kept the running example as simple as possible in order to be able to convey our ideas. However, the meta-models we developed in the context of our tools are more extensive, defining more than a hundred classes. This is due to the fact that the corresponding modeling languages also provide type systems, enable the specification of component properties, and allow us to model the system at different levels of abstraction, like requirements and technical realization [1]. As a consequence, reuse should not only be enabled for components, but also for different artifacts like types, requirements, or hardware components. Therefore, we need a generic reuse concept or pattern that is not only applicable to components, but also to different artifacts.

3 Existing Approaches

Literature on techniques for model reuse is rather scarce, so in this section, we concentrate on describing the two most prominent approaches to integrate reuse

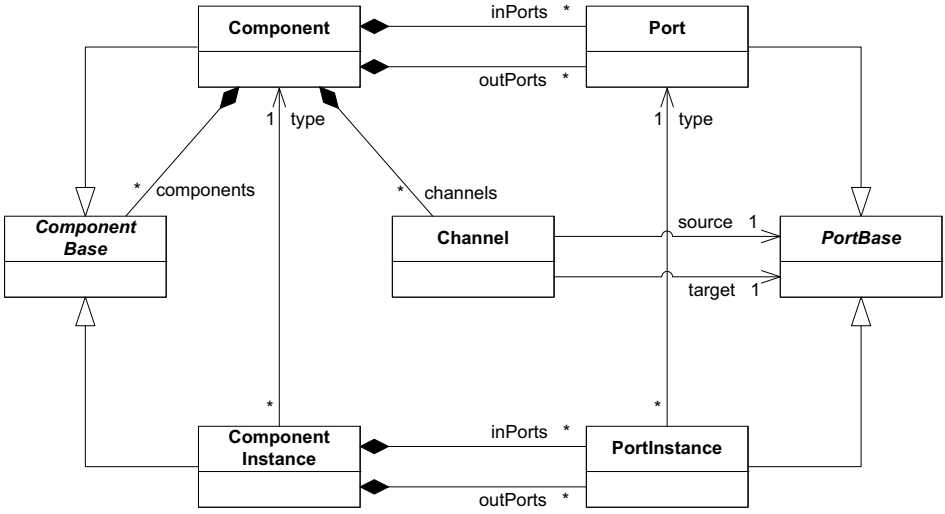

Fig. 2. Introduction of Instance Classes.

into a modeling language. The first approach solves the problem at the level of the meta-model by introducing new classes, whereas the second approach solves it at the level of the meta-meta-model by introducing a generic cloning mechanism. We detail the characteristics of either approach, apply them to our running example, and mention their implications for the meta-model and the modeling tools.

3.1 Introduction of Instance Classes

Reuse can be integrated by introducing classes that explicitly model the instances of types. This technique is presented in [3] where it is applied in a number of domain-specific meta-model patterns.

In the following, the introduced classes are called *instance classes*, whereas the classes of the types are called *type classes*. An association from an instance class to its corresponding type class allows an instance to be aware of its type. This association has to be many-to-one, as there is exactly one type for an instance, and there may be a number of instances for a type. As is depicted in Fig. 2, the class ComponentInstance is introduced to model instances of component types. The corresponding type class Component is accessible from the instance class ComponentInstance through the association type.

To allow an instance to be referenced from other elements, certain child elements of the type have to be instantiated, too. An instance then basically replicates a certain part of the structure of the type which we call *interface* in the following. Additional constraints have to be introduced to the meta-model to ensure that the instance correctly replicates the interface of the type. In our example, the interface of a component also consists of input and output ports. As is depicted in Fig. 2, we thus have to introduce an instance class for ports, namely the class PortInstance, and compositions inPorts and outPorts between ComponentInstance and PortInstance. Furthermore, we have to add constraints to ensure that each component instance

consists of an instance of each of the component's ports.

In order to be able to use instances in the same contexts as types, common super classes for instance and type classes are introduced. These common super classes have to be abstract, as their purpose is to enable instances and types to be used interchangeably. The associations that make up the context in which an element is used have to target these new super classes. In our example, we have to introduce new classes ComponentBase as a common super class of Component and ComponentInstance, and PortBase as a common super class of Port and PortInstance (see Fig. 2). The composition components is re-targeted to the class ComponentBase, so that component instances can also be used to compose components. Furthermore, the associations source and target are re-targeted to the class PortBase, so that port instances can also be connected by channels.

In a nutshell, this pattern requires that the meta-model is extended. For each type class of the type's interface, two new classes have to be introduced: the instance class and a common super class for both instance and type class. Additionally, the structure between the instance classes has to replicate the structure between the type classes. Furthermore, new constraints have to be added to guarantee conformance of an instance to its type. As the meta-model is modified, tools depending on the meta-model, *e.g.*, editors and interpreters, have to be adapted to cater for reuse. This also includes the capability to propagate changes of the interface of the type to its instances. Existing models do not have to be migrated, as the meta-model modification preserves existing model elements. However, future meta-model extensions which enlarge the interface of a type require to introduce new instance classes. When it comes to that, the migration of models is necessary, as additional elements have to be replicated for the existing instances.

We applied this technique to the COLA language [7] which was developed in the context of an industrial project. The instance classes were already integrated, before we started to develop tools for the language. Nevertheless, the tool's implementation grew more complex, as it had to take the additional instance classes into account. A later development step added an additional abstraction level, which required trace links to the existing components. As the composition for sub components was not part of the interface, the sub components of instances are not directly represented as model elements, and trace links thus could not target them. In order to be able to address such an instance anyway, we had to use the following workaround: we reference the instance through the path which leads from the root component to it along the type/instance tree. This technique however decreased the overall simplicity of the meta-model, as these paths have to be kept consistent with the component structure.

3.2 Generic Cloning Mechanism

The previous technique requires only the interface of a type to be replicated. Another technique outlined in [5] is to fully replicate the internal structure of the type. This makes it possible to integrate reuse in a more generic manner at the level of the meta-meta-model. As a consequence, such capabilities have to be provided by

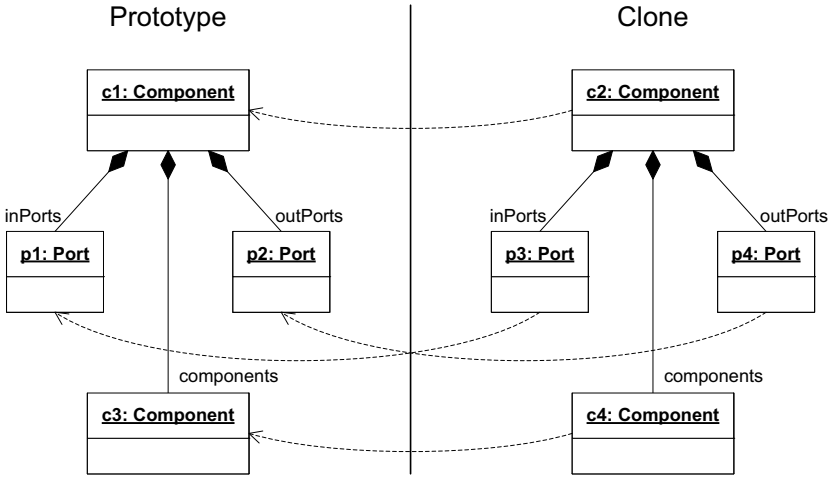

Fig. 3. Generic Cloning of a Component.

the underlying meta-modeling framework. A reference implementation is provided by the authors through GME (Generic Modeling Environment [4]).

A type is basically instantiated by creating a copy of the element. As this technique is known from prototype-based programming languages, a type is called *prototype* and an instance is called *clone* in the following. Due to the meta-modeling framework, the clone is aware of the single prototype from which it is derived. As is depicted in Fig. 3 as a UML object diagram, the component `c1` is reused by producing a clone `c2`. The dashed line indicates that the clone `c2` is associated to its prototype `c1`.

A clone actually is a deep copy of a prototype, meaning that it also replicates the internal structure of the prototype. The clone's children are also aware of the prototype's children from which they are derived, and so forth. Fig. 3 shows that the clone not only copies the component, but also its ports and sub components.

A clone trivially has the same type as the prototype from which it is derived. As a consequence, it can easily be used in the same contexts as the prototype. Therefore, a clone of a component is also of type `Component` and can thus be used to compose other components.

As a clone is aware of its corresponding prototype, changes to the prototype are automatically propagated to the clone. A clone can be modified independently of the prototype – changes to the prototype are then only propagated to the unmodified parts of the clone. In our example, the addition of a port to a component leads to the addition of a clone of that port to all of the component's clones. The clone of the component may be customized by adding a new port.

Due to its genericity, this technique does not require any extensions to the meta-model. However, the meta-modeling framework has to provide the capabilities for cloning model elements. To the best of our knowledge, such a feature currently only exists for GME. As cloning is implemented on the level of the meta-meta-model,

[4] http://www.isis.vanderbilt.edu/Projects/gme/

model elements of all types can be cloned – without restriction. Due to methodological issues, cloning may only be allowed for certain types of model elements. Compared to the previous technique, types have to be completely replicated when instantiating them. In addition, information about which features of clones were overwritten or not has to be maintained within the model. This may excessively increase the amount of information which has to be stored in a model. Furthermore, generic cloning is subject to the following restriction: clones can only be generated from prototypes which do not use cloning in their internal structure. In case this restriction is not enforced, this may lead to clones which do have several prototypes. In Fig. 3, a clone could not be generated from component `c1` if there were a cloning relation between ports `p1` and `p2`, as port `p4` then would have two prototypes – namely ports `p1` and `p3`.

4 Generative Libraries

For the CCTS/AutoFOCUS 3 framework we started thinking about libraries, when the meta-model already consisted of nearly 100 classes and much of the core functionality, such as graphical editors, the expression evaluator, a simulator, and first parts of a code generator, were already implemented, resulting in more than 50.000 lines of source code. Thus the solution had to be minimally invasive, affecting the meta-model and the existing code as little as possible, as resources for going over the entire code base just were not available.

The overall idea was to design a library mechanism, which is orthogonal to the existing meta-model and modeling tools as far as possible. So for the existing code the change should be transparent by providing models that are compatible to those used before. This lead us to an approach similar to [5] where instances are actually full clones of the types. The main difference is that our focus is on a more structured reuse mechanism, where elements may not be arbitrarily cloned, but only dedicated *library elements* may be instantiated. In addition, our approach generalizes cloning in a way that allows us to parametrize instances that are actually generated by the type. In contrast to GME, where cloning is deeply integrated into the meta-modeling framework, our approach can be easily built on top of each existing meta-modeling framework. Our modeling tool for instance is implemented on top of EMF (Eclipse Modeling Framework [5]). It however replicates some of the features of GME in the implementation of the modeling tool. In the following, we outline the overall approach and provide more technical details and implications of our approach.

4.1 The Big Picture

In our setting we differentiate between the library, which contains types, and the importing model, which contains instances of some of these types. The association of an instance to its corresponding type is only performed by name (lazy linkage),

[5] http://www.eclipse.org/modeling/emf/

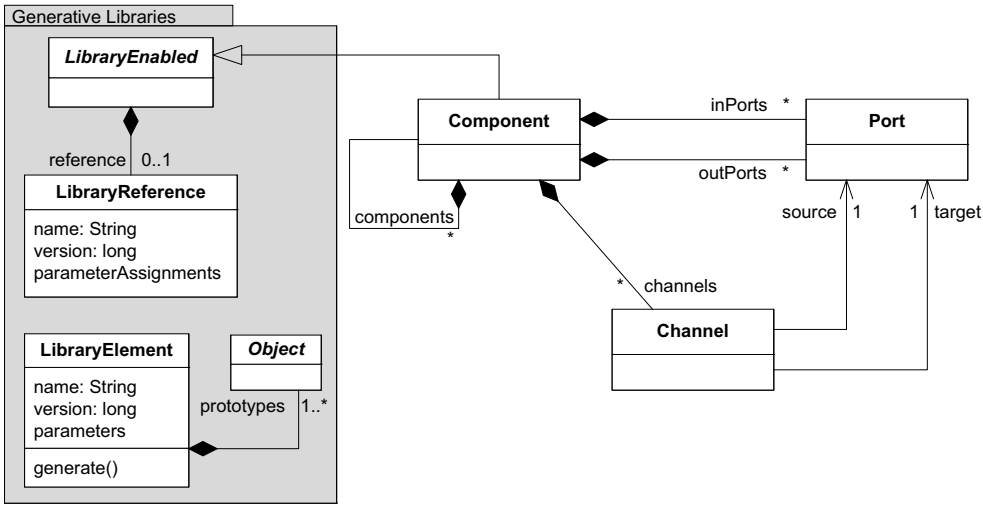

Fig. 4. Meta-model for Generative Libraries.

which allows the implementation of different storage and look-up schemes. For example, we decided to keep libraries in separate files that can be individually imported by other models.

Using our approach, the meta-model grows by three classes[6], no matter for how many existing classes reuse should be enabled. These classes and their usage are shown in Fig. 4.

The abstract class LibraryEnabled is used to mark all meta-model classes which can be used as types. LibraryEnabled introduces a reference, which allows to distinguish "normal" model elements from instances. If the reference does not point to a type in the library, it is a plain element, otherwise the associated LibraryReference stores the name of the referenced type, the version used for the last update, and the parameter assignments used for generating the current instance. In our example, only a Component can be reused via libraries, but for larger meta-models multiple elements can easily be marked to serve as types.

The class LibraryElement is the container for types in a library and defines a name, a version number which has to be incremented with each change of this element, and a set of supported parameters. Most important are the prototypes, which are stored in the LibraryElement. The prototypes composition may include any modeling element (indicated in Fig. 4 by the use of Object which by definition is the transitive super class of all meta-model classes), which can then be used for generating instances of this element. The LibraryElement acts as a factory for a certain kind of class (which is determined by the first prototype in the list) by means of its generate() method which generates an instance based on the parameter assignments. To explain how the meta-model classes are actually used, we explain the three use cases of creating, instantiating, and modifying library elements.

[6] Depending on how the library is organized internally and which mechanism is used for parametrization, there may be more classes for library folders and parameters. The growth in meta-model elements however is constant.

Creation of a library element

The creation of a new type consists of the creation of a LibraryElement, which then is composed of the required prototypes. What information the LibraryElement uses to create an instance from given parameters is up to the implementation. The most basic version just returns a copy of its first prototype, which corresponds to the cloning approach presented in Sec. 3.2. In our tools we also experimented with the inclusion of Groovy [7] scripts, which then generate a new instance using these prototypes based on the given parameters. In our example, this might be the number of ports and the kind of sub components of a component. There are no further limitations concerning the generation, except that a library element always has to return an instance of the same class as generation result (in our case a Component).

Instantiation of a library element

To instantiate a library element for reuse in an existing model, the generate() method of the corresponding element is invoked and the result is inserted into the model. Hence, the generated instance is completely stored in the model, which ensures that the model has the same structure as if built without library support. This is crucial to avoid complicated changes in existing code as mentioned before. The generated element is completed by attaching a LibraryReference which stores the name and version of the library element used for creation. A side effect of the inclusion of plain copies is that the model has all required information even in the absence of the libraries it depends on.

Modification of a library element

The most important reason for building library mechanisms is to support the modification of a library element, which should then be reflected by all of its instances. As we model reuse only by marking instances with their origin and library references are only loosely coupled by their name, we have to deal with certain aspects of reuse in the implementation of our modeling tool. The first part is that the version number of a library element has to be increased with every change to it. As these changes are performed using the modeling tool, this should not be problematic. Using the version number, we can easily identify LibraryReferences, which are outdated (*i.e.*, point to an earlier version of a library element) and thus have to be replaced by an updated version. In our implementation this check is triggered manually by the user to allow full control of the modification of an existing model, as changes in a library are a potential threat to the correctness of a model. However, depending on the tooling infrastructure, such a task could also be performed automatically on certain events. As the prototypes of a library element may again be library instances, the check and update procedure has to be performed in a recursive fashion.

[7] http://groovy.codehaus.org/

The more interesting part is the update of all instances of a library element. Actually, we have to replace a part of the model by a new one, which was generated by the generate() method. A simple exchange of the model parts is not sufficient here, as an instance might include elements which are referenced from other parts of the model (*e.g.*, a channel to one of its ports) or layout and naming information, which we do not want to loose. Our solution uses a modified merge algorithm, which replaces the old model part with the new one, but retains all references to external (relative to the part being exchanged) elements. Using modern reflective meta-modeling frameworks such as EMF, this can be achieved in a generic fashion using only a small amount of code (our implementation has about 300 code lines including comments). However, this piece of code has to be slightly adjusted for the meta-model used, as it has to treat some data differently. For example in AutoFOCUS 3, the name of the top-most element in an instance may be changed by the user and should not be overwritten by a change in the library.

4.2 Technical Considerations and Implications

Using the scheme described so far, most parts of the modeling tool's code can be left untouched. All that is needed is new code for creating and managing the libraries and for handling the update of instances after modifications to their corresponding library elements. There are some additional minor modifications not described here. For example one usually wants to view elements in the hierarchy of a library instance, but it should not (or only in some limited way) be allowed to change them, as those changes would be lost after updating the instance from the library. So the editors have to support (partially) read-only parts of the model. All in all our implementation, which is based on EMF, has about 3.000 lines including the code for managing libraries (folders, etc.) which was not described here. Especially tasks such as updating the version number of a library element upon modification are trivially implemented using recursive listeners which react to all changes to an element *or* its children. The meta-model specific parts of the implementation could be limited to the merge algorithm, though it was kept general enough to make an update only required for severe meta-model changes, such as changing the way layout data is stored. Reuse of the code in other applications is still difficult, as most of it is specific for our modeling tool (*e.g.*, how to switch an editor to read-only mode, etc.).

All in all the implementation and modification effort for our solution was quite manageable, especially as the full (rolled out) model is available and the identification and referencing of individual elements is easy. However, our approach also has some drawbacks. The most obvious is the increased size of the models, as redundant information is stored. While in theory the growth in the number of model elements can be exponential, in our experience the elements being reused are often relatively generic and small, so their number is not a major problem. The larger elements being reused usually solve specialized tasks and thus are used only a couple of times (*e.g.*, a component for monitoring the wheel pressure of a car). Our estimates rather indicate a growth by a constant factor, which depends on the actual model but is in

Criteria	Instance Classes (Sec. 3.1)	Generic Cloning (Sec. 3.2)	Generative Libraries (Sec. 4)
Independence of meta-modeling framework	+	–	+
Growth in meta-model size	–	+	+
Size of models	+	–	–
Deep references	–	+	+
Implementation effort	–	+	0
Reuse strategy	planned	ad hoc	planned
Arbitrary parametrization mechanisms	–	–	+
Automatic update of instances	0/-	+	0/+

Table 1
Comparison of Reuse Approaches

the magnitude of 2 to 10. With the disk space and memory size available today this often is not a big deal, but might be an issue for specific applications or extremely large models.

Another problem with our approach is the slightly concealed reuse structure. While similar or identical parts of the model can easily be determined by looking at the LibraryReferences, their exploitation, *e.g.*, generating the code for these parts only once, can be more complicated than with explicit instance classes. This is especially true in the presence of parametrization.

5 Discussion

To ease the selection of a reuse mechanism and clarify the differences of the presented approaches, in this section we discuss their strengths and weaknesses along several criteria. A summary of this discussion is presented in Tab. 1 which is explained in more detail in the remainder of this section. We want to stress that there is no "best" approach. Depending on the requirements and context each of the presented approaches might be the most suitable. Also the importance of the criteria will be different for every application, so simply "summing up the pluses" is not a valid evaluation. Rather it should be assessed whether there is a hard knock-out criterion (*e.g.*, if the meta-modeling framework is fixed and is not GME, generic cloning might not be a good choice), and then the remaining criteria should be prioritized and evaluated.

Independence of meta-modeling framework

Our first criterion is to what extent the approach is independent of the meta-modeling framework used. Clearly the approach using instance classes can be ap-

plied in the context of any meta-model, as it only uses standard meta-modeling constructs. This also applies to our generative approach, as we do not rely on special features of the meta-modeling framework. Only the implementation might be slightly more complex, if the meta-modeling framework does not support reflection. In contrast, generic cloning is currently only implemented in GME and the extension of a different meta-modeling framework to support generic cloning is a non-trivial task.

Growth in meta-model size

The size and complexity of the meta-model affects the time and effort required for developers to understand and use it. Consequently larger meta-models are more likely to be used in a fashion which was not intended. Additionally, if the meta-model is to be supported by constraints, the number and complexity of these constraints also usually increases with the size of the meta-model. As seen in the previous examples, a number of classes have to be introduced for each kind of type when using instance classes. In contrast, the generative approach requires only a constant number of classes to be introduced, no matter how many classes are marked as types in the meta-model. In generic cloning the same effect is achieved by extensions to the meta-meta-model.

Size of models

As discussed earlier, the size of a model might not be the most critical factor these days, as disk and memory is available at little cost. Still there are applications where the models can reach sizes, which demand a space-efficient representation of the model. These applications might profit from using instance classes, where redundancy in the model is reduced to a minimum. Both of the other approaches store and manipulate a fully expanded model, where all instances are actually full copies of their types. Generic cloning even requires additional storage for tracking changes to the features of all clones.

Deep references

For many applications we have to reference model elements, which are part of an instance. Examples include requirements tracing or deployment models. Creating those deep references is quite easy for generic cloning and generative libraries, as all parts of an instance are realized in the model and thus can be referenced. This is different for instance classes, where these parts are only represented once in the corresponding types. Solutions then either use more complicated constructs for referencing these elements (such as paths along the type/instance tree), or the instances have to be partially realized in the model. In our example, the ports have been replicated for the instance, to allow the channels to connect to them. It is however more complicated to reference the sub components of a component instance.

Implementation effort

Another important factor is the effort required when implementing editors and tools working with these models. The least effort is required with the generic cloning approach when using the GME framework, as for the tools it is completely transparent whether they are working on a prototype or a clone. All cloning and reuse issues are dealt with by GME. This is nearly the same for generative libraries, but the management of types and their instances has to be handled by our tools. As described earlier, the implementation effort for this is still manageable and the editors still do not need adjustment. When using instance classes, the tools dealing with the model, such as editors and simulators, have to cope with the additional instance classes.

Reuse strategy

To some extent, the reuse approach affects the reuse strategy available. We differentiate planned reuse, where elements are explicitly marked or modeled as a type which can be reused by instantiation, and ad hoc reuse, where each element in the model may be used as a prototype. Which one is preferred, depends on the domain used. In embedded systems development, for instance, planned reuse is often favored, as components released in a library might have to be tested and documented more rigorously. In other situations ad hoc reuse might be more suitable, as the modeler does not have to interrupt the modeling flow by deciding which parts to put into a library. Instance classes and generative libraries are more designed toward planned reuse, while generic cloning better supports ad hoc reuse. However, the differentiation between these is somewhat blurry, as of course ad hoc reuse can be restricted by the modeling tools to forge it into a more planned fashion, and automatic creation of library types can be implemented to make the application of planned reuse more "agile".

Arbitrary parametrization mechanisms

Both with instance classes and generic cloning only rather simple parametrization mechanisms are supported. Typically an instance is just a duplicate of the type with possibly some attributes depending on parameters. With our generative approach, where the instance is generated by a method of the library element, any feature of an instance, including the model structure of the instance element, may be affected by parameter choice. This also simplifies the inclusion of different parametrization concepts and thus eases experimentation here.

Automatic update of instances

The crucial aspect when using reuse mechanisms is the update of the instances whenever the type changes. While on the first thought this seems to come at no cost for instance classes, it is actually the most expensive there. In our example, the sub components of an instance are updated automatically, as they are only present in the type, but the port instances always have to be updated, when the ports of the component type change. The situation is similar with generative libraries, but as we

always duplicate the entire instance, the update/merge algorithm can be kept more generic, which dramatically simplifies its implementation. With generic cloning the update is deeply integrated into the meta-modeling framework.

6 Conclusion

This paper provided an overview of different approaches to integrate reuse into models and modeling languages. The focus was especially on the consequences on the meta-model and the implementation of modeling tools. For cases where existing modeling languages are to be extended by library mechanisms, a generative approach was presented, which requires only little modifications to the meta-model and existing code.

To aid in the selection of one of these approaches, we discussed their strengths and weaknesses along several dimensions. None of the reuse approaches clearly dominates the others, but given a specific context and requirements often simplifies the selection. As our discussion reveals, the "classical" solution using type and instance classes often unnecessarily complicates the implementation of tools. In contrast to textual modeling (like source code), controlled cloning seems to be beneficial in many cases.

References

[1] M. Broy, M. Feilkas, J. Grünbauer, A. Gruler, A. Harhurin, J. Hartmann, B. Penzenstadler, B. Schätz, and D. Wild. Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme. Technical Report TUM-I0816, Technische Universität München, 2008.

[2] F. Deissenboeck, B. Hummel, E. Juergens, B. Schaetz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone detection in automotive model-based development. In *Proc. 30th Int. Conf. on Software Engineering (ICSE'08)*, pages 603–612. ACM, 2008.

[3] M. Fowler. *Analysis patterns: reusable objects models*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[4] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proc. of the 31st International Conference on Software Engineering (ICSE'09)*, 2009. To appear.

[5] G. Karsai, M. Maroti, A. Ledeczi, J. Gray, and J. Sztipanovits. Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control Systems Technology*, 12(2):263–278, March 2004.

[6] S. Kelly and R. Pohjonen. Domain-specific modelling for cross-platform product families. In *Proc. Conceptual Modeling - ER 2002, 21st International Conference on Conceptual Modeling (Workshops)*, pages 182–194. Springer, 2002.

[7] S. Kugele, M. Tautschnig, A. Bauer, C. Schallhart, S. Merenda, W. Haberl, C. Kühnel, F. Müller, Z. Wang, D. Wild, S. Rittmann, and M. Wechs. COLA – the component language. Technical Report TUM-I0714, Technische Universität München, 2007.

[8] Object Management Group. Unified Modeling Language, Superstructure, v2.1.2, 2007.

[9] Object Management Group. Business Process Modeling Notation, v1.1, 2008.

[10] Bernhard Schätz and Franz Huber. Integrating formal description techniques. In *Proc. of FM'99, World Congress on Formal Methods*, pages 1206–1225. Springer, 1999.

[11] The MathWorks, Inc. *Simulink 7 User's Guide*, 2008.