Advanced Modelling Made Simple with the Gmodel Metalanguage

Jorn Bettin
Sofismo
Lenzburg, Switzerland
http://www.sofismo.ch/
jbe@sofismo.ch

Tony Clark
School of Engineering and Information Sciences
University of Middlesex, London, UK
http://www.eis.mdx.ac.uk/staffpages/tonyclark/
t.n.clark@mdx.ac.uk

ABSTRACT

Gmodel is a metalanguage that has been designed from the ground up to enable specification and instantiation of modelling languages. Although a number of metalanguages can be used for this purpose, most provide no or only limited support for modular specifications of sets of complementary modelling languages. Gmodel addresses modularity and extensibility as primary concerns, and is based on a small number of language elements that have their origin in model theory and denotational semantics. This article illustrates Gmodel's capabilities in the area of model-driven integration by showing that the Eclipse Modeling Framework Ecore language can easily be emulated. Gmodel offers support for unlimited multi-level instantiation in the simplest possible way, and any metalanguage emulated in Gmodel can optionally be equipped with this functionality.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.12 [**Software Engineering**]: Interoperability

General Terms

Binding times, Denotational semantics, Domain analysis, Graphs, Instantiation semantics, Metamodels, Model-driven integration, Model theory, Modularity, Multi-level modelling, Scope management, Value chain modelling

1. INTRODUCTION

In order to increase awareness about the role that domain-specific modelling languages can play in capturing, preserving, and exploiting knowledge in virtually all industries, it is necessary to:

1. *Reach a consensus on fundamental values and principles for designing and using domain-specific languages*

2. *Progress towards interoperability between tools*

– KISS Initiative, 2009 [4]

The development of Gmodel relates to the second objective of the KISS initiative [3], and builds on the KISS results that have been achieved in 2009 [4]. In particular, Gmodel represents an attempt to provide explicit tool support for the full set of KISS principles:

1. *The DSL must be meaningful to users*

2. *The DSL should be cognitively efficient*

3. *The DSL should have multiple notations where necessary*

4. *DSLs should offer mechanisms for modularising and integrating models*

5. *The DSL should be supported by appropriate tooling*

6. *There must be an economic imperative for the development of a DSL*

7. *The DSL must not be polluted with implementation features*

8. *Model processing must always be based on a formal DSL definition*

9. *DSLs should be kept small through modularisation and integration*

Since the design of Gmodel rests on mathematical concepts from model theory and from the theory of denotational semantics, Gmodel can tap into established mathematical terminology, and the target audience for Gmodel includes modellers in all disciplines. Consistent with denotational semantics and with the third KISS principle, Gmodel completely separates the concern of representation from the concern of naming. This means that in contrast to most programming language specifications, the specification of Gmodel does not include a text-based concrete syntax.

The authors of Gmodel believe that modelling has the greatest value when performed by domain experts, and if modelling language design takes into account established domain notations. The challenge consists in providing a metalanguage that enables the most experienced domain experts to define the notation for modelling in their field, whilst at the same time providing tool support for enforcing (and ideally

guaranteeing) the adherence to KISS modelling language design principles.

This paper starts with a brief introduction of relevant terminology, and then presents the Gmodel metalanguage in the context of model-driven interoperability based on practical examples:

1. Introduction of Gmodel kernel concepts
2. Outline of Gmodel's contribution to model-driven interoperability
3. Representation of the Eclipse Modeling Framework Ecore language in Gmodel, and outline of the design of a bidirectional bridge between the two technologies
4. Description of advanced modelling techniques for scope management, modularisation, and interoperability
5. Comparison of Gmodel with other technologies
6. Conclusions

2. TERMINOLOGY

Modellers are not in the business of inventing new terminology, they are in the business of identifying concepts and links between concepts that are useful for a particular community of people – usually scientists or professionals in a particular field. This approach to modelling is consistent with the Oxford American dictionary definition of modelling:

to model devise a representation, especially a mathematical one of (a phenomenon or system)

2.1 Model Theory

The mathematical definitions from model theory make use of the concepts of sets and graphs, and provide a mathematical basis for reasoning about models:

structure A structure **A** is a set that contains the following four sets

1. A set called the *domain* of **A**, written as dom(**A**)
2. A set of elements of **A** called *constant elements*, each of which is named by one or more *constants*
3. For each positive integer n, a set of n-ary relations on dom(**A**), each of which is named by one or more n-ary *relation symbols*
4. For each positive integer n, a set of n-ary operations on dom(**A**), each of which is named by one or more n-ary *function symbols*

signature The signature of a structure **A** is specified by giving the set of constants of **A**, and for each separate $n > 0$, the set of n-ary relation symbols and the set of n-ary function symbols – The symbol L is used to represent signatures and languages; if **A** has a signature L, **A** is also called an *L-structure*.

Beyond these two definitions, model theory defines the following concepts that every modelling language designer should be familiar with: *substructure, term, formula, variable, language, cardinality, sentence, theory, model* [7].

This terminology and the associated mathematical theory have heavily influenced the design of Gmodel.

2.2 Denotational Semantics

The second source of influence on Gmodel is denotational semantics, in particular the concepts of *semantic domain* and *semantic identity* [11].

One advantage of using established mathematical terminology to describe Gmodel is a low risk of terminological confusion with concepts from the Meta Object Facility (MOF), a popular metalanguage that is steeped in object orientation, and with concepts from related implementations such as the Eclipse Modeling Framework Ecore language. This benefit immediately becomes apparent when discussing the representation of Ecore in Gmodel. A second advantage of using the above terminology is the ability to reason about Gmodel in mathematical terms, without the need for any linguistic gymnastics.

2.3 Natural Language and Exchange of Artefacts

In addition to mathematics, Gmodel terminology draws on concepts that have shaped the development of natural language, and the way in which humans perform work and exchange artefacts – including abstract ideas. In relation to the latter, and in accordance with the second KISS principle, the design of Gmodel takes into account human cognitive abilities and limitations [10].

language artefact A *container of information* that:

1. is *created by a specific actor* (human or a system)
2. is *consumed by at least one actor* (human or system)
3. represents a *natural unit of work* (for the creating and consuming actors)
4. *may contain links to other language artefacts*
5. *has a state and a life-cycle*

model artefact A language artefact that meets the following criteria:

1. It is *created with the help of a software program* that enforces specific instantiation semantics (quality related constraints)
2. The information contained in a model artefact *can be easily processed by software programs* (in particular transformation languages)
3. *Referential integrity between model artefacts is preserved at all times* with the help of a software pro-

gram (otherwise the necessary level of completeness and consistency is neither adequate for automated processing nor for domain experts making business decisions based on artefact content)

4. *No circular links between model artefacts are allowed at any time* (a prerequisite for true modularity and maintainability of artefacts)

5. The *life-cycle of a model artefact is described in a state machine* (allowing artefact completeness and quality assurance steps to be incorporated into the artefact definition)

instance A set that *seems to contain* one and only one element at any given point in time from the *view point* of a specific *actor*

instantiation function A function that returns an *instance* – sometimes instantiation functions are also called *concretisation functions*

visibility Visibilities are links between model artefacts that set the *architectural context* for artefact *producers* by declaring the model artefacts that can be used as inputs for the creation of specific kinds of model artefacts

producer An *actor* that creates *language artefacts*

consumer An *actor* that consumes *language artefacts*

value chain A value chain consists of actors (systems and humans) that consume artefacts as inputs and produce derived artefacts

3. THE GMODEL KERNEL

The Gmodel kernel [figure 1] is a semantic domain consisting of a set of semantic identities that reify the concepts of ordered pair, ordered set, and graph – the latter consisting of a set of vertices and a set of edges. To facilitate extensibility and multi-level instantiation, the encoding of the Gmodel kernel is entirely expressed in Gmodel semantic identities, and each semantic identity in the kernel is defined as an instance of itself, and as a sub set of the next simpler semantic identity in the kernel.

The generic term to refer to any semantic identity that is expressed in Gmodel is the *set*. The simplest semantic identity is the *ordered pair*. Ordered pairs are used to define *ordered sets* and *graphs*. Much of the power and simplicity of Gmodel has its origin in the specific encoding chosen for graphs. Instead of consisting of a set of vertices and a set of edges, a Gmodel graph is encoded as a set of vertices and several complementary ordered sets of *links*:

edges Links between two *sets* with a dedicated *edge end* for each connected set

super set references Directed links from a *sub set* to a *super set*

visibilities Directed links from one *sub graph* to another *sub graph*

edge traces Directed links from one *edge* to another *edge*

The Gmodel *vertex* and all four types of *links* are encoded as *sub sets* of *graph*. In order to serve as a metalanguage, edge ends are decorated with variables for *minimum cardinality* and *maximum cardinality*, as well as variables that represent the direction of *navigability* of edges and a notion of *containment* relating to the connected set.

3.1 Instantiation

A modeller may use the *instantiation* function of Gmodel kernel to create representations of vertices and links. Since vertices are encoded as a sub set of graph – and hence enable the representation of nested abstractions, vertices are well positioned to serve as the unit of modularity in Gmodel. Using the terminology introduced above, vertices play the role of *model artefacts*, and in the context of Gmodel (modelling), are simply referred to as *artefacts*.

Links between artefacts are also encoded as a sub set of graph, and therefore are also capable of representing nested abstractions by containing sets of vertices and sets of links. Links between two artefacts are always contained in the artefact that contains the first of the two artefacts connected by the link, which is one of the constraints that allows Gmodel to fulfil the fourth criteria of the definition of model artefact – effectively enforcing much stronger rules regarding modularity than the minimum expectations set by KISS principles (4) and (9).

The most powerful feature of Gmodel instantiation is the ability to decorate any Gmodel artefact with *instantiation semantics* (or *concretisation* semantics) relating to representations of less abstract (or more *concrete*) sets, such that the artefact becomes *instantiable*. The instantiation semantics available in the Gmodel kernel are specified via the variables for *cardinalities*, *navigability*, and *containment* that are part of all *edge ends* of Gmodel edges. Thus, on the one hand, by excluding any circular links between artefacts, Gmodel imposes heavy constrains on the models that can be created, but on the other hand, Gmodel allows an unlimited degree of freedom with respect to the number of instantiation levels.

Gmodel does not mandate a layered metamodel architecture. Our modelling experience in software intensive industries has taught us that the model pattern known as the *power type pattern* in object orientation occurs pervasively in highly configurable systems. The power type pattern is a technical kludge that forces the fragmentation of semantic identities, and it clearly demonstrates the limits of the object oriented paradigm – which is currently still treated as dogma by many software engineers. By allowing multi-level instantiation, the need for the power type pattern is eliminated, and the fragmentation of semantic identities can be avoided.

3.2 Surface notation

The name Gmodel is motivated by the *graph* concept, and all notations for visualising graphs are good candidates for a concrete syntax for Gmodel artefacts. In contrast, purely text-based representations are only practical for representing Gmodel artefacts with certain characteristics, such as

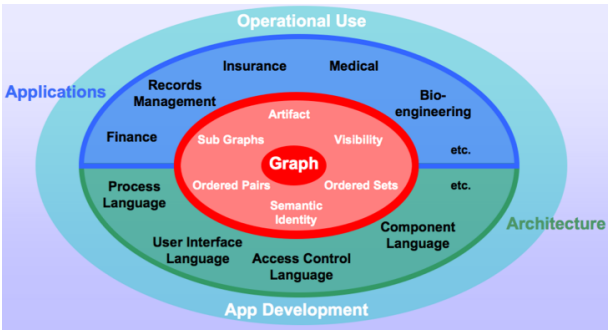

Figure 1: The Gmodel kernel within a typical usage context

artefacts with a low ratio of edges to vertices. Additionally, given that the main target audience for Gmodel consists of modellers in general – as opposed to software engineers with a strong preference for working with formal text-based languages, there is no urgent need for developing a human readable purely text-based syntax.

The Gmodel open source project has no intention of reinventing XML or burdening the world with yet another XML-based but not-quite-human-readable syntax. The Gmodel API can easily be used to build graphical editors for Gmodel artefacts that are complemented with appropriate form-based representations of variables and their values. At this point in time Gmodel provides two complementary graphical surface notations for visualising model artefacts. A generic graphical editor that allows artefacts to be created and modified is currently under development.

3.3 Model artefact storage

Gmodel internally uses a serialisation format that is not intended for human consumption, and it provides a binding of this serialisation format to relational database technologies. In particular Gmodel fulfils criteria (3) of the definition of model artefacts, and provides explicit support for the semantics of *unknown* and the semantics of *not applicable*. As needed, the serialisation format can be bound to alternative persistence mechanisms such as file systems, object databases or cloud database technologies.

In order to work with model artefacts, Gmodel includes a repository API that currently offers basic artefact search functionality, which will be significantly enhanced in future releases.

3.4 Interoperability mechanisms

There are two main ways of achieving interoperability between Gmodel and other modelling technologies. This article focuses on the level of profound semantic interoperability with other metalanguages that can be achieved by making use of multi-level instantiation to emulate "foreign" technologies. Gmodel also offers an alternative for partial and superficial interoperability via file based information exchange. Out of the box Gmodel includes integration with the Eclipse integrated development environment, and with the openArchitectureWare Xpand template/transformation engine, putting text or code generation at the user's fingertips.

4. CONTRIBUTION TO MODEL-DRIVEN INTEROPERABILITY

To a significant degree the development of Gmodel was motivated by a lack of adequate technologies for formal modelling beyond the realm of software engineering and programming languages, and by a lack of interoperability between existing tooling for domain-specific modelling languages. Gmodel simplifies model-driven interoperability in the following areas:

1. Modularity – The implementation of the artefact concept prevents users from constructing circular dependencies between modules. In contrast to other technologies, Gmodel allows the modelling of links between primary model artefacts and derived artefacts [figure 6], which amounts to an in-built infrastructure for orchestrating model transformations.

2. Simplicity – Since the Gmodel kernel treats links between concepts as first-class constructs, all kinds of graph structures (including undirected graphs) can be represented without compromise, and in the preferred terminology of the user. As a result, representations of modelling languages within Gmodel tend to be highly compact [figure 6], and the complexity of any required model transformations is reduced accordingly.

3. Multi-level-modelling – Gmodel is not limited to the four layered metamodel architecture. This opens up new approaches with respect to interoperability [5] [2], since types – and therefore interoperability patterns, can be encoded in Gmodel to any level of complexity. As illustrated in this article, multi-level instantiation is a prerequisite for emulating "foreign" modelling technologies. We are not aware of any other multi-level modelling technology that is ready for industrial use.

4. Scope management – Gmodel has an explicit feature for scope management that is universally available within all modelling languages expressed in Gmodel [figure 9]. This gives designers of modelling languages and system architects an unprecedented amount of control over the artefacts that language users can instantiate. In the experience of the authors, such functionality is essential for managing the dependencies between languages and between components in large-scale software intensive systems.

5. Separation of the concern of modelling from the concern of naming – Right down to the core Gmodel functionality is expressed in semantic identities, and these identities can be referenced from as many representations (models) as needed. In practical terms this allows Gmodel to incorporate custom terminology and jargons at all meta levels.

6. Portability – In contrast to many other modelling technologies Gmodel makes no assumption about the implementation and legacy technologies that modellers are going to drive from their model artefacts. The Gmodel kernel is highly portable. It is articulated using the concepts presented in this paper, and makes use of the Java programming language to bootstrap the nine kernel concepts of *ordered pair, ordered set, graph, vertex, edge, edge end, super set reference, visibility*, and *edge trace* – but without exposing the Java type system in the core API, whilst restricting internal use of Java types to a handful: `boolean`, `int`, `List`, `Iterator`, `UUID`, and `String`.

5. EMULATING ECORE IN GMODEL

Gmodel clearly distinguishes between *semantic domains* and *models*. The former simply contain sets of *semantic identities*, whereas the latter contain *representations of semantic identities* from the view point of a particular *actor*.

5.1 Representing the Ecore metamodel

In Gmodel no model can be constructed without referencing elements in the relevant underlying semantic domains.

5.1.1 Defining the Ecore semantic domain

In Ecore the most generalised element is the EObject, and all other elements are part of a generalisation/specialisation hierarchy that starts with EObject. To represent Ecore in Gmodel, the first step consists of instantiating the semantic domain `EcoreDomain`, which contains all the semantic identities that appear in Ecore [figure 2]. This step will be perceived as somewhat unusual by all those who are only familiar with the definition of text-based languages using EBNF-style grammars; as the concern of representation and the concern of naming are one and the same in such specifications.

The number of semantic identities required to represent Ecore is significantly larger than the number of elements that appear in the Ecore generalisation/specialisation hierarchy. Every instance of an EDataType, every instance of an EReference, every instance of an EAttribute, etc. that occurs in the encoding of Ecore in itself requires a corresponding semantic identity. Loosely speaking, everything that has a name in the encoding of Ecore in itself corresponds to a semantic identity.

5.1.2 Representing the representation of Ecore

To prepare for the representation of Ecore in itself (the metametamodel level in the classical four layered metamodel architecture) in Gmodel, we instantiate a *model artefact* (with meta element *vertex*) based on the semantic identity `Ecore` that has been defined as part of the EcoreDomain in the previous step. Loosely speaking we now have an empty model artefact called `Ecore`.

We can then proceed to add *contained artefacts* to the `Ecore` artefact that correspond to the Ecore generalisation/ specialisation hierarchy that starts with EObject. Once this is done we can represent the entire Ecore generalisation/ specialisation hierarchy in the `Ecore` artefact using *super set references* as shown in figure 3, and we can represent all instances of EReferences in the `Ecore` artefact within Gmodel as illustrated in figure 4.

Lastly we add all relevant *variables* to the elements of the `Ecore` artefact, making use of appropriate *semantic identities* from the `EcoreDomain`.

The whole process of representing Ecore in Gmodel is straightforward modelling in Gmodel, and requires no coding in a programming language.

5.2 Representing Ecore models

The representation of Ecore models (the metamodel level in the classical four layered metamodel architecture) in Gmodel follows the same pattern as the representation of Ecore in itself in Gmodel. First, appropriate semantic identities must be defined, and then the `Ecore` model artefact can be instantiated to obtain an empty model artefact. Note that above we instantiated a vertex to obtain a model artefact with the `Ecore` semantic identity, and now we are instantiating this model artefact.

Just as above, the next step consists of adding contained artefacts to the model artefact, this time however the meta elements of the contained artefacts correspond to Ecore concepts. Up to this point there is nothing special about using Gmodel. We could turn the table and proceed with very similar steps in Ecore to obtain a reasonable representation of Gmodel – "reasonable", because Ecore actually lacks one instantiation level to provide a precise representation of Gmodel *edges*. But instead of delving into the encoding details of Gmodel edges, the following step in encoding Ecore models is straightforward to follow, and clearly illustrates where multi-level instantiation plays a critical role.

In Gmodel we can proceed to represent all instances of EReferences as demanded by the Ecore model we are emulating, and we can use the edges that Gmodel uses to represent EReference instances to record the cardinalities pertaining to the *instantiability* of the model artefact. In a metalanguage without multi-level instantiation we would already have hit rock-bottom at this point. We would have been able to express links between elements (which, depending on the metalanguage, may be called "references","association", "relationships", "connections", "edges" or similar – the name is immaterial), but we would not have been able to decorate these links with cardinalities etc., which constitute essential instantiation semantics for the next level of instantiation or concretisation.

5.3 Representing instances of Ecore models

Given the explanations above, it is obvious how to proceed to instantiate Ecore models (the model level in the classical four layered metamodel architecture) in Gmodel such as the

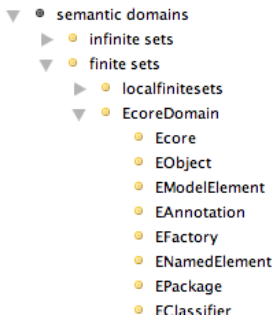

Figure 2: The Ecore semantic domain

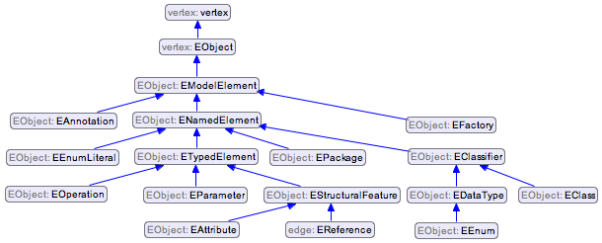

Figure 3: Encoding of Ecore super types

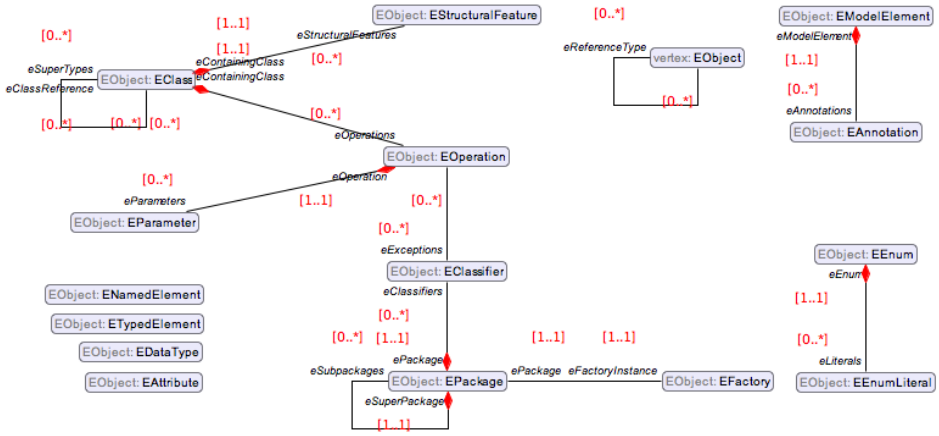

Figure 4: Encoding of Ecore references

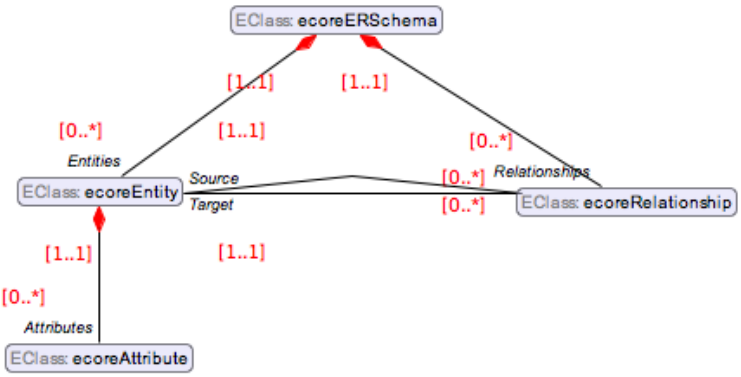

Figure 5: Encoding of an entity relationship modelling language in the Ecore emulation

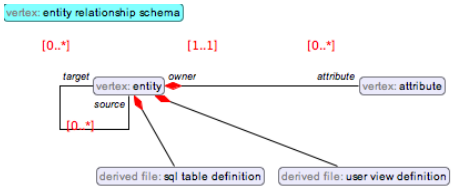

Figure 6: Native encoding of an entity relationship modelling language in Gmodel

example shown in figure 5. The comparison with figure 6 illustrates the complexity introduced one the one hand by the emulation, and on the other hand by the Ecore encoding of links between concepts in the form of EReferences.

5.4 Representing instances of instances of Ecore models

In Gmodel there is no reason to stop modelling at the "model" level. If the modeller has invested in decorating a model artefact with instantiation semantics, Gmodel is capable of applying these semantics – regardless of the level of instantiation or concretisation [figure 7].

In practical terms multi-level instantiation allows the modeller to instantiate operational data right down to the *concrete* level (the instance level in the classical four layered metamodel architecture) – where Joe Bloggs owns life insurance policy number 123456 [figure 8].

Given that industrial-strength relational database technology is the default storage format used by the Gmodel repository, navigating and maintaining large databases or data warehouses is simply a matter using the Gmodel repository for navigation, and of using Gmodel's instantiation function.

5.5 Interoperability between Ecore and Gmodel

With the native encoding of Ecore emulated by Gmodel artefacts, building a bi-directional bridge between the two technologies has become a trivial task. The Ecore API can be used to systematically read EMF models (at the metamodel level and the model level in the classical four layered metamodel architecture), and the retrieved in-memory representations can be mechanically mapped to corresponding in-memory representations in the Ecore emulation within Gmodel.

Gmodel is a technology that allows the construction of model-driven systems on a new scale, whereas EMF Ecore is a technology with an established user base and a vast array of useful transformation and generator components that facilitate the binding to popular Java implementation technologies. A bridge between Ecore and Gmodel can be driven by an event-based mechanism to create dynamic interoperability between the two technologies, opening up interesting avenues for model-driven systems that exploit the strengths of both technologies.

6. ADVANCED MODELLING TECHNIQUES

Modularity and scope management go hand in hand. One without the other is of very little value.

6.1 Scope management via visibilities

Gmodel requires users to be explicit about scope. A model artefact may not reference any element in other model artefacts unless these artefacts have been declared to be *visible* from the first artefact [figure 9]. In contrast to most programming languages, declarations of visibility are not part of an artefact, but they are part of the *parent artefact* in the so called *artefact containment tree.*

The parent artefact has the responsibility of providing the *architectural context* for all the artefacts that it contains. The authors of Gmodel consider it to be good modelling practice to associate every artefact with a producer, and to identify and name the binding time that is associated with the instantiation of an artefact. Experience from many large-scale software system development initiatives has consistently confirmed the usefulness of this approach to system analysis and modularisation.

The encoding of Ecore in Gmodel required the declaration of a small number of visibilities, but there are much better practical examples that can be used to demonstrate the value of scope management via visibilities – a topic that goes beyond the scope of this paper.

Visibilities offer significant value to intensive users of EMF, as Ecore lacks a corresponding facility. By switching from the native implementation of Ecore to the Gmodel `Ecore` emulation, EMF users gain access to the use of visibilities, and hence obtain a powerful tool for actively managing/restricting the dependencies in large-scale Java component architectures.

6.2 Applications of multi-level instantiation

6.2.1 The bottomless pit of abstractions

Gmodel incorporates the insight from experienced modellers that there is no absolute rock-bottom *concrete* level of models. Life insurance policy number 123456 only looks like an instance from the view point of the average policy holder. From the view point of the insurer a specific version of the policy that is active for a certain interval is a more appropriate perception of *instance.* If, in 2020, Joe Bloggs decides to shift his entire life into n virtual worlds (given the track record of software technology, who would want to put all eggs in one basket), his view point will shift. Life insurance policy number 123456 in Second Life may be considered to be one instance, and the corresponding policy representation in Third Life may be considered to be a different instance – perhaps the currency in which premiums are being paid is different in each of the virtual worlds.

6.2.2 Value chain modelling and mass customisation

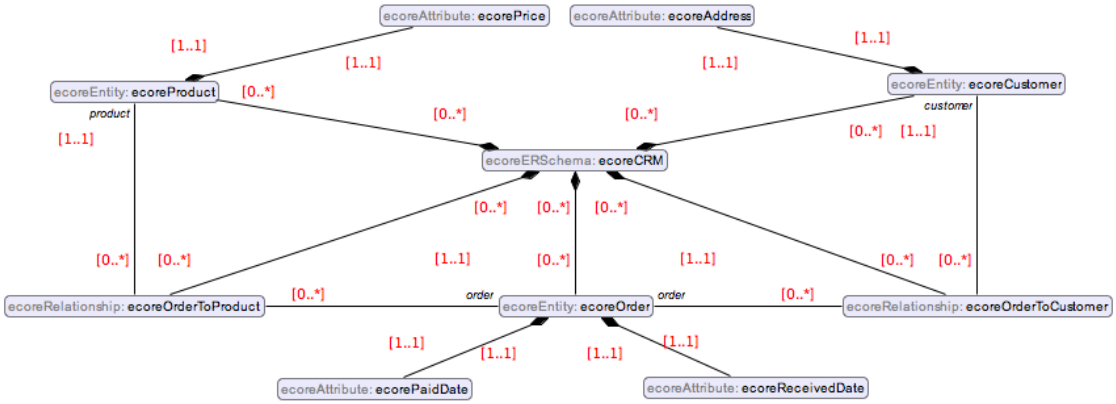

Figure 7: Snippet from an entity relationship model of a CRM application

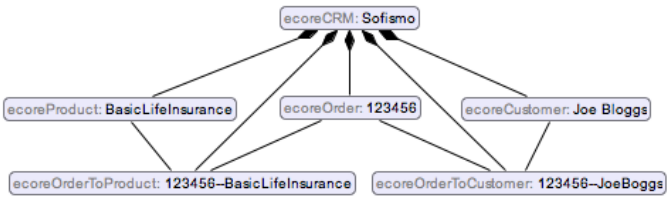

Figure 8: Joe Bloggs' life insurance policy number 123456

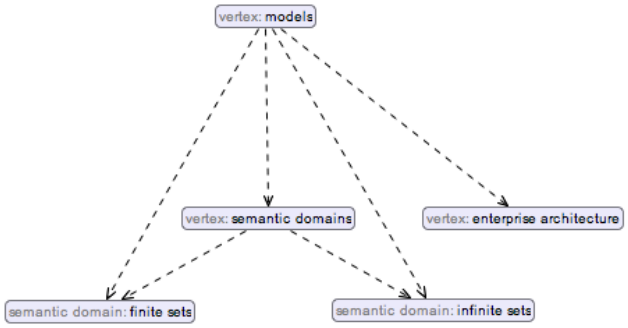

Figure 9: Example of visibility declarations

If the above sounds far fetched, analysing the typical evolution of technology products over a period of several years provides further motivation for multi-level instantiation. Since the 1970s software has been used as a tool to not only automate industrial production, but also to extend the degree to which technology products can be configured and customised without having to resort to manual manufacturing techniques. Mass customisation has become commonplace in many industries.

The evolution of a product over longer stretches of time can be modelled as a series of instantiation levels. Adding a new set of configuration options equates to adding additional *variables* to an artefact that used to be perceived as an instance. What used to be called a product morphs into a *product line*, and the new products are the instances of the product line, where each of the variables take on concrete *values*. The view point of the customer usually remains unaffected, she still buys instances of a product.

Within a non-trivial value chain, the variables associated with a product line tend to be replaced by concrete values in a series of stages, so called *binding times*. Each binding time is associated with a specific actor that is responsible for making *decisions* regarding the values relating to a specific set of variables. In our experience multi-level instantiation is by far the simplest modelling technique for representing non-trivial value chains.

The alternative of using a purely object oriented design, in combination with the classical power type pattern, leads to system designs that are much more complex and much less maintainable than they could be. In particular the traditional distinction between design-time and run-time is a dangerous over-simplification that distracts from the need of proper *value chain analysis* (also known as *domain analysis* in the discipline of software product line engineering).

6.3 Applications of denotational semantics

Since all model artefacts in Gmodel are constructed from semantic identities, and since semantic identities are the only Gmodel elements that have names, semantic identities offer a one-stop-shop for dealing with all aspects of naming. This greatly facilitates any required translation between different terminologies, and it even enables users to replace the names of the semantic identities in the Gmodel kernel. If a user prefers to call a vertex a node, or if she prefers to rename *TRUE* to *FALSE* and *FALSE* to *TRUE*, so be it. The role of modelling is representation and not naming.

Separating the concern of *modelling* from the concern of *naming* adds value precisely because good terminology *is* so important. Each pair of collaborating actors in a value chain tends to have a preferred terminology or *jargon* for their specific interactions, and such jargon is often a valuable tool for *disambiguation*.

Without the systematic use of semantic identities, establishing interoperability across an entire value chain is significantly complicated. Names end up being used in the definition of protocols and artefacts, and the reliability of links between the participants in the value chain and communication across the links suffers accordingly.

It is worthwhile to note that semantic identities are not only applicable at the atomic level to define identities such as *TRUE* and *FALSE*, but are just as applicable to statements such as *minimum cardinality = 1* or to aggregates such as the entire `Ecore` model artefact.

7. OTHER TECHNOLOGIES

The level of interoperability between current domain-specific modelling tools is comparable to the level of interoperability between CASE tools in the 90s. To increase the popularity of model based approaches, this needs to change. The assumption that all parties in a global software supply chain will use identical tooling is simply not realistic.

7.1 Research prototypes

We are aware of at least three modelling technology prototypes with some form of multi-level instantiation capability [1], [8], [9], [6]. It would be extremely interesting to compare the design of Gmodel with the design of these prototype technologies.

7.2 Eclipse Modeling Framework Ecore

In this paper we have illustrated how Gmodel can be used to emulate the Ecore technology, and conversely we have highlighted some of the limits of Ecore, in particular the lack of support for multi-level instantiation.

7.3 MetaEdit+

MetaEdit+ is mature metamodelling and modelling environment that compares favourably with the Eclipse Modeling Framework. In particular the metametamodel of MetaEdit+ is simpler than the metametamodel used by Ecore, without any sacrifice in expressive power. But just as Ecore, MetaEdit+ follows the four layered metamodel architecture dogma and does not offer multi-level instantiation. As a result, MetaEdit+ runs into the same limitation that Ecore runs into when attempting to emulate "foreign" modelling technologies.

Similar to Gmodel, MetaEdit+ relies on database technology rather than a file system for the storage of model artefacts, enabling modellers to build large-scale model-driven systems, but without explicit scope management facilities.

7.4 Unified Modelling Language tools

The main target audience of UML consists of software professionals who have an interest in visualising code, especially object oriented code. Most UML tools only offer very limited – if any – functionality for instantiating models that users have created. Since UML is based on the Meta Object Facility (and on Ecore or similar implementations), UML tools are affected by the kinds of limitations discussed in this paper in relation to Ecore.

7.5 Programming languages

There are several programming languages that offer multi-level instantiation, and there are also a number of programming languages that are based on denotational semantics, such as LISP or REBOL. Whilst these language have expressive power that is comparable to Gmodel, they don't offer the *limitations and constraints* that have consciously been built into Gmodel.

Programming language designers approach language design from a view point that differs significantly from the view point of a modelling language designer.

1. A programming language is designed to be executable on a specific platform. The platform represents the *solution space*, and the implementations of programming languages are optimised with respect to using the resources offered by the platform.

2. Since most programming languages are general purpose languages, they have to offer features that cover the needs of a big range of different users. As a result programming languages offer many features that are not strictly needed by the majority of users. These features lead to additional degrees of freedom in solution designs, and consequently lead to variations in implementation that are induced by personal design preferences of individual software engineers. In the small this may not matter, but in the large these variations are known as spurious complexity.

3. A modelling language is designed for the representation of specific kinds of problems. As outlined in this article, problem spaces are best modularised along the lines of the actors that participate in a value chain, and each actor must be equipped with modelling languages that have a clear focus on the specific context and view point – all other details must be abstracted away. The result is a design force that pulls in the opposite direction of the design force that drives the development of most programming languages. The most valuable modelling languages are not only domain-specific, they are company specific.

Gmodel is a metalanguage that strives to provide expressive power in those areas that matter to modellers, and at the same time it strives to restrict those expressive powers that may lead to non-maintainable artefacts.

8. CONCLUSIONS

Although Gmodel is a brand new metalanguage, it embodies the collective lessons from many experienced modellers. The specific constraints that have been built into Gmodel have a track record of many years in industrial practice. Up to now best practices for scope management and modularity had to be applied manually, in the form of conventions. This worked up to a point, but it posed limits to the scalability of modelling technology in large environments.

Without appropriate tool support, designing and maintaining advanced model-driven systems requires a large number of highly skilled modellers and system architects, and often the required level of expertise is simply not available. We hope that Gmodel offers the missing stepping stone that allows a much larger group of organisations to reap the benefits of formal modelling, by significantly reducing the number of concepts and technologies that a designer of modelling languages needs to be familiar with, and by offering features – such as multi-level instantiation – that lead to simpler and clearer designs. Interoperability with EMF Ecore as outlined in this article is currently being refined, and a bi-directional bridge between Gmodel and Ecore will be a feature in an upcoming release of Gmodel.

No modelling tool can ever replace the need for domain analysis, but Gmodel is ideally positioned to record the results of domain analysis. On the one hand Gmodel provides domain-specific modelling support for all participants in a value chain, and on the other hand it serves as a front end for model transformation and code generation technologies that allow models to be glued to existing technologies and legacy systems.

9. REFERENCES

[1] Colin Atkinson, Matthias Gutheil, and Bastian Kennel. A flexible infrastructure for multilevel language engineering. *IEEE Trans. Softw. Eng.*, 35(6):742–755, 2009.

[2] Jorn Bettin and Tony Clark. Gmodel, a language for modular meta modelling. In *Australian Software Engineering Conference, KISS Workshop*, 2009.

[3] Jorn Bettin and Tony Clark. The knowledge industry survival strategy initiative (kiss), 2009.

[4] Jorn Bettin, William Cook, Tony Clark, and Steven Kelly. Knowledge industry survival strategy (kiss): fundamental principles and interoperability requirements for domain specific modeling languages. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 709–710, New York, NY, USA, 2009. ACM.

[5] Tony Clark, Paul Sammut, and James Willans. Applied metamodelling: A foundation for language driven development, 2008.

[6] Tony Clark, Paul Sammut, and James Willans. Superlanguages: developing languages and applications with xmf., 2008.

[7] Wilfrid Hodges. *A shorter model theory*. Cambridge University Press, New York, NY, USA, 1997.

[8] A. Laarman. An ontology-based metalanguage with explicit instantiation, March 2009.

[9] Alfons Laarman and Ivan Kurtev. Ontological metamodeling with explicit instantiation. In M. van den Brand, D. Gaševi?, and J. Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 174–183, Heidelberg, January 2010. Springer Verlag.

[10] Tomasello M, Carpenter M, Call J, Behne T, and Moll H. Understanding and sharing intentions: The origins of cultural cognition. *Behavioral and Brain Sciences, 28, 675 - 691*, 2005.

[11] David A. Schmidt. *Denotational semantics: a methodology for language development*. William C. Brown Publishers, Dubuque, IA, USA, 1986.