Verified Visualisation of Textual Modelling Languages

Fintan Fairmichael[1] and Joseph R. Kiniry[2]

[1] University College Dublin, Ireland
[2] IT University of Copenhagen, Denmark

Abstract. Many modelling languages have both a textual and a graphical form. The relationship between these two forms ought to be clear and concrete, but is instead commonly underspecified, weak, and informal. Further, processes and tool support for modelling often do not treat both forms as first-class citizens, instead choosing to favour one as the "real" representation and the other as a derivable representation.
As textual and graphical forms have their individual strengths and weaknesses, ideally one should be able to view and edit a model in whichever form is most desirable at the moment. Furthermore, we should be able to do so without having to worry about semantic differences between what is seen in a graphical view versus what is seen in a textual view. If we are to develop tools that allow dual-editing—simultaneous editing of both the textual and graphical forms—then it is essential that their relationship is clearly and precisely defined.
This paper details a formal relationship between the textual and graphical forms of a high-level modelling language called the Business Object Notation (BON). We describe the semantics of the graphical and textual representations and the relationship that holds between them. We also formally define a view on an underlying model as an extraction function, and model diffs as a means of tracking changes as a model evolves. This theoretical foundation provides a means by which tools guarantee consistency between textual and graphical notations, as well shows how to efficiently perform model updates, reason about model views, and interpret properties between modelling perspectives.

1 Introduction

A wide variety of modelling languages and tools are now available to developers. For those languages that have both graphical and textual representations, it is essential that these are consistent such that the text that appears in the textual form and the dots, lines and shapes that are drawn for the graphical form are semantically equivalent or consistent.

The specification of the relationship between the textual and graphical forms of a modelling language should be more than the reference implementation of a tool that converts from one to the other.

Edits to either representation change the underlying model in some fashion, and a corresponding edit should be applied to the other representation. For instance, when we draw a new class `A` in our graphical editor, a class `A` should also be added to our text, and vice versa. It is crucial that both forms are updated accurately and kept in sync. If no synchonisation guarantees are made, then the model a user is viewing in one or other representation may not be accurate, leading to a plethora of potential issues as development continues.

Frequently, each form represents a different subset of the overall model. For example, the Object Constraint Language (OCL) for UML has no graphical representation. OCL annotations are simply encoded in UML diagrams using unstructured *notes* or semi-structured *stereotypes*. Similarly, most graphical forms contain some layout information that is not expressed in the corresponding textual notation. Since we are concerned with the interrelationship and interactions of the two formats, in these cases we simply consider the subset of each form that is relatable, as the matter of maintaining this extraneous information is one of careful engineering.

By formalising the textual and graphical models of BON, and the relationship between them, we provide a precise notion of consistency between these two representations. By also formalising the notion of a view on a model, we are able to examine some common views, how they relate to the original model, and their translation from one representation to another. Before looking in detail at this formalisation, we first discuss our modelling language of choice, BON.

1.1 BON

The Business Object Notation (BON) was developed from within the Eiffel community, back in the late 1980s and early 1990s, as a descriptive method that addresses both analysis and design issues [7]. In fact, much of its syntactical style is inherited from the Eiffel programming language.

BON has two *levels*, which are loosely referred to as the informal and formal levels. Diagrams at the informal level are comprised of natural language, written in a highly structured manner. At the formal level we have dependent types, including behavioural contracts on features[3] (pre/postconditions) and classes (invariants). Inheritance (with generics and multiple superclasses) and client relations (associations and aggregations) are also expressed at the formal level.

Classes are hierarchically arranged in clusters, which are structures that simply group together one or more classes and/or clusters. Clusters are similar to packages in Java, except without the additional effects on visibility that packages exert. A well-formed model in BON must have a single top-level cluster—the *system cluster*.

Classes and clusters are described in the static part of a BON model, but there are also dynamic diagrams that allow one to describe events and scenarios over the static system. In this work we are concentrating on the static and formal

[3] Features in BON can be thought of as methods or fields, but are more general than these and can potentially be refined to either when creating an implementation from the model.

parts of BON, although broadening it to cover the entire modelling language is a natural extension.

BON models have both a textual and a graphical form, with a one-to-one structural mapping between them. The BON authors also describe interesting views on models, through the use of *compressions*—representing a set of graphical or textual elements with a simpler element [14]. This is standard practice in many graphical systems as a user zooms out, and similar techniques are employed through outline views and folding for textual editors. However, when dealing with a modelling language, the meaning of these simpler elements is important in the context of the elided elements that they represent. It is made all the more interesting in BON, as we can write and display relationships between clusters, as an abstraction from the underlying classes upon which the relationships really exist. We discuss this further in Section 3.3.

```
static_diagram EXAMPLE
component
  cluster EXAMPLE_COMPONENTS
  component
    class A
      feature
        name: STRING
        set_name: Void
         -> n:STRING
         require X.is_valid_identifier(n)
        end
    end
    class B inherit A end
    class C inherit B end
    class D inherit B end

    B client UTILITIES.X

  end

  cluster UTILITIES
  component
    class X
      feature
        is_valid_identifier: BOOLEAN
          -> id:STRING
    end
  end
end
```

Listing 1. A static diagram in Textual BON

Listing 1 shows an example of a textual BON model, which contains several classes linked through inheritance and a single client relation. The same model represented in graphical BON is shown in Figure 1.

1.2 Tools

BONc We have previously developed a parser, typechecker and documentation generator for BON, the BONc tool.[4] It is open source, and available as a com-

[4] Available from http://kind.ucd.ie/products/opensource/BONc/

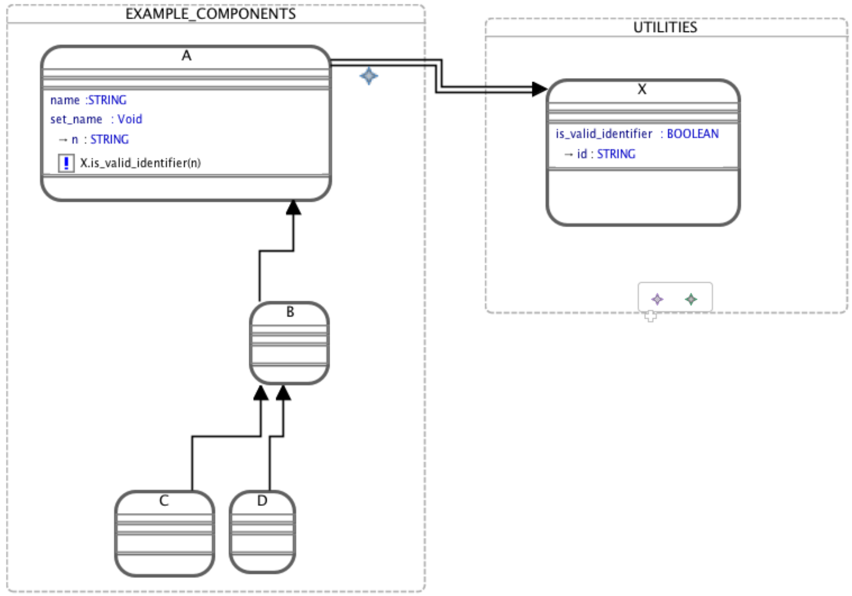

Fig. 1. Graphical BON.

mand line tool or as an Eclipse plugin. We have used BONc as a foundation for a number of projects, including a model-code consistency checker and the BON IDE.

BON IDE We have recently developed a prototype visual editor for BON, built on top of the Eclipse Platform, leveraging the Eclipse Graphical Modeling Project (GMP) tools [11].

A BON metamodel is defined using the Eclipse Modeling Framework Project (EMF) Ecore metamodel. Defining the BON metamodel as an Ecore model empowers the use of the Eclipse GMP and Graphical Editing Framework (GEF) to aid in the development/generation of tools for manipulating graphical BON. The culmination of these efforts is a graphical editor for BON—the BON IDE. The screenshot previously shown in Figure 1 is taken of the BON IDE in action. The BON IDE checks the validity of a graphical BON model through constraints expressed in the Ecore metamodel, with additional validity expressions written in the *Check* language (part of the Model To Text (M2T)[5] Project).

The remainder of this paper is organised as follows. The next section describes the formalisation of graphical and textual BON models, the translation between these, as well as views on a model. Section 3 discusses some uses of our setup, in particular tracking model evolutions and keeping models synchronised. We next discuss related work, before finally drawing conclusions about the current state and future directions of this work.

[5] http://www.eclipse.org/modeling/m2t/?project=xpand

2 Model Formalisation

We have mechanically formalised our textual and graphical models in higher-order logic using the PVS Specification and Verification System [8]. Much of the formalisation is presented here in standard mathematical syntax (i.e., the reader need not be familiar with PVS to understand our theory), but several small examples of PVS are given as well to concretise our mechanisation. The motivated reader is welcome to download the full mechanisation via our website.[6]

PVS provides an interactive proof checker for proving the correctness of theorems as well as type-correctness of the input specifications. By mechanising our theory in PVS we are able to assert desirable properties of our specification in a manner that is easily independently verified by others.

2.1 Graphical Formalisation

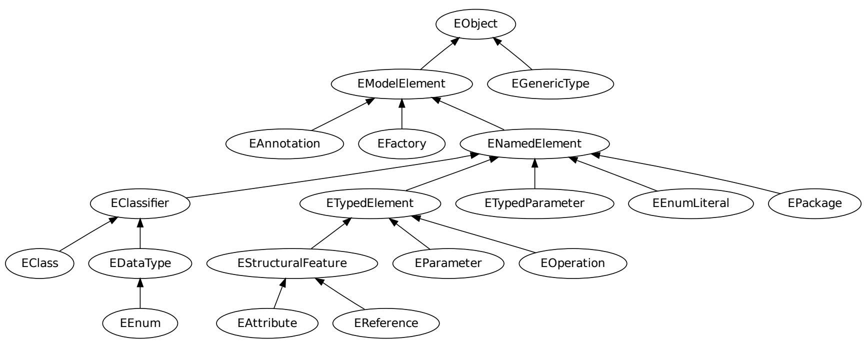

Fig. 2. Ecore type hierarchy.

Due to our use of the Eclipse Modeling Framework [13] it was first necessary to describe the core Ecore types, and their type hierarchy. The relevant parts of this type hierarchy are depicted in Figure 2.

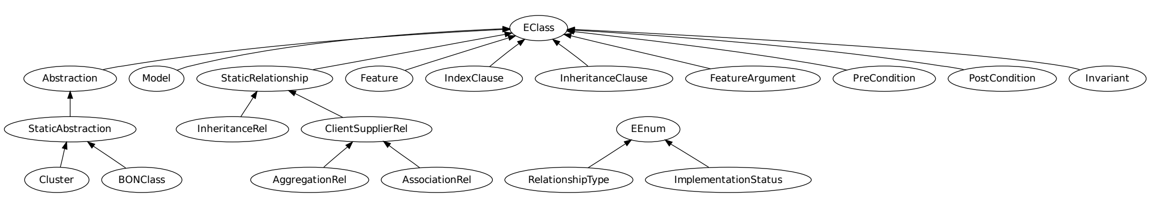

Fig. 3. BON graphical model type hierarchy.

[6] Available from http://kindsoftware.com/documents/mech_theory/verified_vis.html

BON types are defined as subtypes of the relevant Ecore constructs, summarised in Figure 3. A formal definition of several elements of this type hierarchy are necessary. Types that are not mentioned below have mechanisations but are irrelevant to the focus of this paper.

Definition 1 (Basic Constructs) A *system* is a BON model instance. A *class* is a named concept or idea in a system. A *cluster* is a named set of *classes* and sub-*clusters*. Every system has exactly one cluster denoted as the *system cluster*. □

We denote classes with the metavariable $\mathcal{C}$, clusters with $\mathcal{D}$, and the system cluster as $\mathcal{S}$. Functions on classes and clusters exist for extracting the relevant attributes (e.g., identifier, list of features, etc.), but are not discussed further here.

Informally, a BON model consists of a set of classes, a set of clusters, and a set of relations between these classes and clusters. The general idea is that inheritance relations define the subtyping relations between classes, cluster relations give the clustering structure of the system, and client relations state dependencies between the classes in the system. Each of these constructs, in turn, is formalised in the following.

The inheritance, cluster, and client relations of a system are all dependently typed directed graphs ("*digraphs*" within PVS). An inheritance graph is more precisely a *tree* of `BONClass`es, rooted with the `ANY` class.[7]

Definition 2 (Inheritance Tree) An *inheritance tree* is a (directed) tree whose vertices are elements of the type `BONClass` and whose root is the `BONClass` `ANY`. □

This definition is expressed in PVS in the following fashion.

```
ANY: BONClass
inheritance_tree: TYPE =
  { t: Tree[BONClass] | root?(t)(ANY) }
```

This PVS specification states that the constant `ANY` is of type `BONClass` and the type `inheritance_tree` is a tree of `BONClass`es whose root vertex is `ANY`. Note that `inheritance_tree` is a simple dependent type.

A *cluster graph* is also a tree, with the system cluster at its root. Since `StaticAbstraction` is subtyped by both the `Cluster` and `BONClass` types, we thus use `StaticAbstraction`s as the vertex type in our cluster tree.

Definition 3 (Cluster Tree) A *cluster tree* is a directed tree CT of `Static-Abstraction`s where the root of CT is the *system cluster*, and every non-root vertex in CT has exactly one *cluster* as its parent. □

The last condition within this definition is necessary to insist that only clusters may contain other static abstractions (i.e., classes must be leaf vertices within the cluster graph), and every static abstraction is directly contained within one cluster (except the system cluster).

[7] `ANY` is the top-most type in BON.

Definition 4 (Client Graph) A *client graph* is a directed graph of `BONClass`es. Each edge in the graph is a client relation. □

Note that, as no further constraints are placed on the *client graph*, cycles are permitted—as would be expected for client relations.

A *BON model* is comprised of an inheritance tree, a cluster tree and a client graph. Additionally, all classes in the system must appear as vertices in both the inheritance tree and the cluster tree, and no other classes may appear in the client graph. The metavariable used to denote a *BON model* is $\mathcal{M}$.

Definition 5 (BON Model) A *BON model* is a record $[IT, CT, CG]$ where:

- IT is an *inheritance tree*, CT is a *cluster tree*, and CG is a *client graph*,
- the set of `BONClass`es that are vertices in CT must be equivalent to the set of vertices in IT,
- and the set of `BONClass`es that are vertices in CG must be a subset of those appearing in CT/IT. □

This definition is expressed within PVS by declaring the record type and a predicate that determines if such a record meets the latter two criteria above. The `BON_model` type is then the subset of these records that satisfies this predicate.

```
BON_model_rec: TYPE =
  [# it: inheritance_tree,
     ct: cluster_tree,
     cg: client_digraph #]

BON_model?(m:BON_model_rec): bool =
  vert(m`it) = class_vert(m`ct)
 AND
  subset?(vert(m`cg), vert(m`it))

BON_model: TYPE = (BON_model?)
```

2.2 Textual Formalisation

BON textual models are represented as a *type context*. Classes are stored in the type context in a partial function from a `FormalClassType` to a `FormalClassDefinition`. The function is partial, as it only returns a class definition when such has been given in the system and stored in the type context. Asking for the definition of a class type that is not present in the type context returns `bottom`.[8] The metavariables $\mathcal{CT}$ and $\mathcal{CD}$ denote `FormalClassType`s and `FormalClassDefinition`s, respectively.

A `FormalClassType` contains the identifier that denotes the class in question and a list of generic parameters for the type it represents. The *type* function extracts the type from a given class definition: $type : \mathcal{CD} \to \mathcal{CT}$.

[8] In a partial function in PVS the function returns `bottom` when the function is undefined for the input.

Definition 6 (Class Definition Map) A *class definition map* $\mathcal{CDM}$ is a partial function $def : \mathcal{CT}' \rightarrow \mathcal{CD}$ where:

- $\mathcal{CT}'$ is a subset of $\mathcal{CT}$ and
- $\forall ct' \in \mathcal{CT}' : type(def(ct')) = ct'$. □

The second condition in this definition ensures that class types are only mapped to class definitions that actually have the type in question.

Clusters are similarly stored in a partial function from `FormalClusterType` to `FormalClusterDefinition`. We omit the precise definition for brevity.

`TypeRelationPair`s are used to represent the various relations between types in a system with subtypes `InheritanceRelation`, `ClusteringRelation` and `ClientRelation`. In our PVS mechanisation `TypeRelationPair` is an *abstract datatype* with these three subtypes.[9]

Using all of this mathematical infrastructure, one can now define *BON textual type contexts*.

Definition 7 (Type Context) A BON textual *type context* is a record

$$[CDM, CLDM, S, R]$$

where CDM is a class definition map, $CLDM$ is a cluster definition map, S is the system cluster type, and R is a set of `TypeRelationPair`s. □

As usual, the metavariable Γ denotes a BON textual type context, and the sentence $\Gamma \vdash \diamond$ indicates that Γ is well-formed.

Type contexts are created and reasoned about with a standard suite of operations that guarantee they remain well-formed and consistent. As is normal, our formalisation of type contexts only permits types (i.e., classes and clusters) to be *added* to the context, not *removed*, at this time.

2.3 Relating Textual and Graphical Formalisations

Next comes the meat of the matter: relating the theory of *textual* BON to the theory of *graphical* BON.

The key idea here is that one incrementally defines a semantics-preserving bijective function, called a *model interpretation*, that maps from a well-formed textual type context to a well-formed graphical type context. As the function is bijective, it implicitly defines an equivalence class between (pairs of elements in) the contexts, and hence between contexts' contents (classes and clusters). Moreover, since the interpretation is semantics-preserving, any property proven about constructs at one end of the interpretation can be "pushed" through the interpretation to its other end.

[9] Abstract Datatypes in PVS automatically generate additional axioms. This includes an axiom that each type must be one of its subtypes (inclusive) and an axiom that the subtypes do not overlap (disjointness). These properties are especially useful for case analysis.

Providing the details of this interpretation is beyond the scope of this paper. The interested reader should obtain our PVS mechanisation and look, in particular, at the theory `viz_to_typesystem`. This theory contains about a dozen critical lemmas, of which we have mechanically proven half at this time, and sketched out proofs of the other half by-hand.

Commonly, one only defines this kind of interpretation as a model relation; but, as we have carefully crafted the theories at both ends of the relation, we have also defined "executable" interpretation functions that go "both ways," from graphical to textual and vice-versa.

Mathematically, this is unsurprising and not necessarily terribly useful. But practically, the existence of these functions is extremely useful, as any well-formed type context created with a tool realising either side of the relation (i.e., whether a developer draws a picture or writes a textual specification that describes a system) can be interpreted into the dual semantic domain. Performing the defined conversion in PVS involves applying the relevant functions to rewrite a model from one representation to another. As such, the conversion scales to large models, although the number of intermediate steps in the conversion may grow with the model size. A textual specification is "rendered" into a graphical specification, and it is guaranteed that all properties of the textual specification are captured in the graphical specification. Likewise, a graphical specification is "pretty-printed" into a textual specification, and the same semantic guarantees hold.

Finally, the relation itself is useful as it permits one to check that two specifications, graphical and textual, are *consistent* with each other and if one is a *refinement* of the other.

Definition 8 (Consistent Models) A (graphical) BON model $\mathcal{M}$ is *consistent with* a (textual) type context Γ *iff* there exists a quintuple of maps between classes, clusters, systems, and relations that are bijections. We write $\mathcal{M} \cong \Gamma$ if this is the case. □

Definition 9 (Refinements as Submodels) A model $\mathcal{M}$ is a *refinement* of a type context Γ *iff* there exists a quintuple of maps between classes, clusters, systems, and relations from $\mathcal{M}$ to Γ are surjections. We write $\mathcal{M} \sqsupset \Gamma$ if this is the case. This definition is symmetric. □

Now that we can compare textual and graphical specifications, and understand what it means to refine between them, we must reflect on the kind of operations that we can perform to either end of the interpretation, and what each of those operations means in the dual domain. The first operation of interest is *views*, as it is not always the case that one wishes to render an entire graphical model.

2.4 Views

We can informally define a view as the display of a subset of the elements (classes, clusters, relations) in a model. In order to formally define a view, we first define the type of a (generic) model view. A *model view* is a "slice" of a model amenable to visualisation. As our models have three components—inheritance and cluster hierarchies and a client relation graph—a view slices through all three components.

Definition 10 (BON Model View) A *BON Model View* for a *BON Model* $[IT, CT, CG]$ (recall Definition 5) is a record $[IT_v, CT_v, CG_v]$ where IT_v is a (possibly empty) subgraph of IT (*resp.* CT_v and CT, and CG_v and CG). $\square$

The metavariable we use for the *BON Model View* type is $\mathcal{V}$, and we denote the corresponding relation as $\circledcirc : \mathcal{M} \times \mathcal{V}$.

A relation is expressed in PVS in one of several ways. We have chosen to formalise this relation as a (curried) function with a codomain of type boolean.[10] The definition is based upon a case distinction on all subtypes of $\mathcal{V}$. The function type definition in PVS is shown below.

```
BON_model_view?(m:BON_model)(v:BON_model_view_rec): bool
```

Using a curriable representation enables us to define a type for all legal views of a given model. We do so by defining a dependent type of a view function that takes a `BON_model` and returns a valid view for the given model.

Definition 11 (The View Type) The *View Type* $\mathbb{V}_M$ is a function type dependent upon a model M and is of type $m : M \rightarrow (v : V | m \circledcirc v)$. $\square$

Such a dependent type is realised in PVS by the following type declaration. Note that `(BON_model_view?(m))` is the type of all views on `m`.

```
view: TYPE = [m: BON_model -> (BON_model_view?(m))]
```

3 Uses

3.1 Model Evolution

As models change over time, keeping track of those changes and maintaining the consistency of the graphical and textual forms is crucial. We will first define a few relevant functions on graphs, leading up to a definition of a *diff* on a pair of BON models.

Definition 12 (Graph Merge) A *graph merge* is a function $gm : G \times G \rightarrow G$ that produces a graph that contains a union of the vertices and edges from the two input graphs. $\square$

[10] In PVS, the declaration of a function F of type $A \times B \rightarrow C$ is denoted as `F(A,B):C`, in which case any use of the function must include all parameters; whereas a curried function type $A \rightarrow (B \rightarrow C)$ is denoted `F(A)(B):C`. The latter allows for partial application yielding a new function of type $B \rightarrow C$. As relations can be represented as $A \times B \rightarrow \mathbb{B}$, if we use the curried form we can, through partial evaluation, obtain a predicate on the type B. Consequently, since PVS uses predicate subtyping in HOL, this partial application denotes a new subtype of B.

The difference of two graphs is a new graph with vertices as the difference of the two input graphs' vertices, and edges as the difference of the two input graphs' edges.

Definition 13 (Graph Difference) A *graph difference* is a function $gd : G \times G \rightarrow G$, such that the resultant graph contains all the edges and vertices that are contained in the first of the input graphs, but not in the second. □

A *model difference record*, MD, is a record $[IT_d, CT_d, CG_d]$ where IT_d is a directed graph of `BONClass`es, CT_d is a directed graph of `StaticAbstraction`s and CG_d is a directed graph of `BONClass`es. Model difference records are used to store the output of a model difference.

Definition 14 (Model Difference) The *model difference* function $\mathit{difference} : M \times M \rightarrow MD$ creates a record containing the graph differences of each of the constituent graphs of the input models. □

A *model difference* gives us what was present in the first model and not in the second—i.e., the elements that are removed when translating from the first model to the second. However, we also need to know what was present in the second model but not in the first (the elements that were "added").

Definition 15 (Model Diff) The *model diff* function for input models m_1, m_2 produces a record $[\mathit{add}, \mathit{rem}]$, where $\mathit{add} = \mathit{difference}(m_2, m_1)$ and $\mathit{rem} = \mathit{difference}(m_1, m_2)$. □

This is akin to the *diff* file comparison utility that represents the changes between two files through the parts that are added and the parts that are removed. We now define a diff application function for BON models that "applies" a diff to a given model.

Definition 16 (Model Diff Application) The *model diff application* function $\mathit{apply\_diff} : M \times MD \rightarrow M$ creates a new model produced by adding all elements in the `add` field of the model diff and removing all elements in the `rem` field. □

Technically the output of a model diff application is a BON model record, as the diff used might produce an invalid BON model. In PVS this means that we produce a `BON_model_rec` that does not necessarily satisfy the predicate `BON_model?`. We could of course define a predicate that tells us if a given input model and model diff would produce a valid model through *apply_diff*, but this is not likely to be more useful than simply applying the diff and then seeing if the result is a valid model.

One essential property of the *apply_diff* and *diff* functions is that

$$\mathit{apply\_diff}(m_1, \mathit{diff}(m_1, m_2)) = m_2$$

That is, if we take the diff of two models, and apply this diff to the first of those models we should produce the second model. This property is intuitive from the understanding that a diff describes the changes that must be made to the first input model to produce the second.

It is possible to define a bijective translation of the BON graphical model diff to a (similarly defined) textual diff that operates on the BON textual model, allowing updating of the textual model from the graphical model (and similarly the graphical model from the textual model) by computing the diff for the changes, translating this diff and applying it to the textual model. This removes the need to translate the full model every time we want to synchronise our two model forms.

3.2 Chasing Refinements

Given the relation and functions defined in Section 2.3, there are a number of interesting properties we can "push" between worlds. In essence, much like in theoretical physics and other branches of mathematics involving symmetries, sometimes it is easier to state or reason about a property in one theory than another.

We propose taking advantage of this situation in a few novel ways, though we summarise only a few here due to space reasons. Note that we have *not* mechanised any of these ideas as of yet, nor have any been implemented in our toolset in a fashion that reflects our formalisation.

The Right Tool Principle Certain activities, like brainstorming or sketching out an architecture, are often better accomplished in one tool (e.g., a graphical model editor, like the BON IDE) than another. Since an interpretation exists between domains, generating a complete, human-readable, consistent textual specification from a "sketch" of a system is trivial.

We call this idea "the right tool" principle—one should not be prevented from using the right tool for the job and one should be able to leverage such work in both domains.

Difficult Properties At first blush, the idea of a *well-formed* graphical model seems simple enough. But we found encoding such in a framework as complex as the EMF+GEF to be fraught with peril. On the other hand, defining such a property in the textual model is a straightforward proposition. Consequently, we do not bother attempting to specify complete and complex well-formedness conditions on a graphical model, we simply translate from the graphical to the textual and check well-formedness there.

Axiomatic Properties Axiomatic properties like well-formedness are delicately realised in textual and graphical tools. When *should* such properties hold, and when *must* they hold? In our experience, too many tools, especially those with formal underpinnings, improperly force consistency-like properties.

For example, many modern compilers are able to successfully compile erroneous programs by making assumptions about developer intent and performing runtime error correction of the input program. Of course, they issue warnings in such cases, but overall this behavior makes the tools easier to use and more robust. The Java compiler included with the Eclipse IDE is an excellent example. Eclipse can perform type-completion and summarise a class in the outline view even if the program code in the editor is not type-correct.

As another example, while using a tool like the BON IDE, often it is not just *useful*, but is *necessary* to draw an *inconsistent* or *ill-formed* model. Perhaps such happens during brainstorming; perhaps in a few more steps the model will be consistent again; perhaps the designer is modelling an existing system which, by virtue of its very definition, is inconsistent in the first place.

By formally defining such properties on either side of the interpretation and understanding precisely when reasoning infrastructure depends upon their validity, we are able to characterise how and when to enforce them. Within PVS we realise such a theoretical trick by putting properties that are *strictly* necessary for reasoning within the definitions of our predicate types, while those that are *optional* are encoded as axioms or lemmas. If a definition of a formula F that is used to reason about a model, or proofs relating to such a formula, does not involve a given axiom or lemma P, then we know F and P to be *independent*, and as such is *optional*.

3.3 Views

The simplest and least interesting views are of course the empty view, where none of the model elements are displayed, and the full view, where all elements of the model are displayed. This section discusses a couple of more interesting views commonly provided for textual models, and explores what should happen to relationships involving elements that are hidden in a view (i.e., present in the original model, but not in the view).

Outline View Outline views of textual modelling languages typically give a hierarchical view of the principal elements in the model. It is clear that an outline view is indeed a view, under our definition, since the hierarchies in question are present in the original model. Outline views are less common for graphical editors, but the principle remains the same and can be provided in a similar manner.

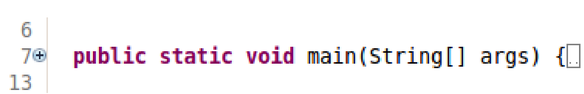

Fig. 4. Code folding in the Eclipse Java editor.

Folding Another common feature with modern textual editors is folding. This is where a large, multiline element can be collapsed in the view to a single line. For instance, a class definition that normally spans dozens of lines can be collapsed to just one line. Typically a small graphical widget in the left margin of the line denotes the folding state of an element, as well as to function as a button for the fold and unfold operations. An example of code folding in the Eclipse Java development tools (JDT) source editor is shown in Figure 4. Here a method body (lines 8–13) has been hidden from view, with graphical artifacts to indicate this to the user.

Collapsing or restoring a foldable element are relatively straightforward alterations to the view on a model. When an element is collapsed, the view has been altered so that sub-elements in the heirarchical structure of the original model are ellided from the view.

Relationships Involving Hidden Elements Folding is related to compression in BON, as previously discussed in Section 1.1. When one performs folding, compression, or some other view adjustment where model elements are being hidden as a means of providing a simplified or abstracted view of the system, it raises an interesting question as to what should happen to the relationships that exist between model elements that are hidden. In the case where model elements are hidden because they are not relevant, then of course no relationships involving these elements should appear in the view.

Consider the specific example of views to visualise the dependencies that exist in a system. Client relationships are expressed between classes, and in our definition of a view no relationships can appear involving a class that is not present in the view—since a relationship is an edge in a graph, we cannot have an edge involving a non-existent vertex. However, a view that showed the dependencies that exist between clusters that did not contain the clutter of the underlying classes would be useful. The client relationships are "pulled up" in the clustering hierarchy to appear at the cluster level.

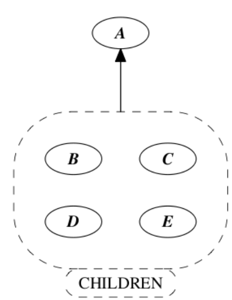

Fig. 5. Graphical BON compression example (from [14]).

It is this type of view that the BON authors were keen to support. Figure 5 shows a simple example of an inheritance relationship that has been abstracted from the underlying classes `B`, `C`, `D`, `E`, to their containing cluster `CHILDREN`.

How to amend our definition of a view of a model, such that it can accommodate this "pulling up" of relationships to higher-level abstractions, is an ongoing research question for us.

4 Related Work

There is a great abundance of related work on modelling languages and tools that deal with textual and graphical representations, so we will only briefly mention a few of those that are relevant.

Lancaric et al. developed a case tool for BON that supported drawing and editing of graphical BON class diagrams (static) and dynamic diagrams [6,9]. The graphical representation they used largely followed that originally set out by Walden and Nerson. Skeleton code generation is possible to the target languages of Eiffel, Java with JML, and textual BON. Importing existing BON models is not supported, so round-tripping during development is not possible.

Goguen's early work on formal semiotics [4] and Feferman's work on reasoning with diagrams [3] has been inspirational to our work. Both researchers argue, in essence, that only by connecting the formal world (of which computing, and thus system design, is definitely one of the most complex examples thereof) with a grounding in social reality via semiotics, or the general theory of signs, can one effectively communicate complex constructs in effective ways. Moreover, proofs appealing to diagrams and other semiotic constructs are ubiquitous in geometry, number theory, and especially category theory.

UML-B is a profile of UML, which defines a subset and specialisation of UML that has a mapping to, and is therefore suitable for, translation into the (textual) B language [12]. Although an equivalency of UML features to B structures is defined, this work is differentiated by the fact that UML-B operates across two very disparate modelling languages.

The Alloy Analyzer [5] tool contains a visualiser for graphically displaying Alloy models. However, the graphical form is not editable so synchronisation of the graphical with the textual formats is simply a matter of updating the visualiser with the current version of the textual model. UML2Alloy [2] defines a profile of UML (and OCL) suitable for translation to Alloy, and translates an XMI representation (exported from ArgoUML) of a UML model from this profile to an Alloy textual model for analysis. Again, this process is in one direction—from graphical UML to textual Alloy.

Our description of model diffs is related to the work of Alanen and Porres [1], although our definitions are simpler as we operate on graph structures and treat a change to a class or cluster (a vertex/node) as the removal of the old version and the insertion of the new.

Modern Eclipse platform projects provide powerful tools for creating textual and graphical editors. The Xtext project focuses on supporting the development of textual editors and integrates with the Eclipse Modeling Framework. Thus, a modelling language textual editor built on top of XText can integrate with graphical modelling tools built using the Eclipse Graphical Editing Framework. Model edits are represented through Eclipse model transactions and these can be used to keep the graphical and textual models synchronised.

Paige and Ostroff previously developed a formally specified metamodel for a subset of the BON language [10]. Two versions of the metamodel were produced: the first written in BON itself, and the second formalised in the PVS specification language. Their metamodel was used as a starting point for our own Ecore-based metamodel.

5 Conclusions and Future Work

We believe that formalising the textual and graphical sides of a modelling language and their interrelationship helps to bring clarity to the semantics of the language and the representations employed therein. Semantically consistent translations and updates to the model representations are essential to provide high reliability on the validity of models that have been automatically updated by tools.

Our tool support is at an early stage; in particular, the BON IDE cannot yet be used to describe any of the dynamic parts of the BON modelling language. Translation between the graphical and textual models as described here has been started, but not completed.

There also still remains a great deal to be completed in the formalisation. The work described in this paper only deals with the static elements of formal BON, but providing full coverage is both desirable and necessary to reason about complete BON models.

This approach can be generalised and applied to other modelling languages, such as UML with OCL. The complexity is likely to rise with the size of the language, however, and the utility will be reduced if the textual and graphical forms are only loosely integrated.

6 Acknowledgements

Thanks to Ralph Skinner for his work on the BON IDE. This work is supported by the EU Artemis project CHARTER.

References

1. Alanen, M., Porres, I.: Difference and union of models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003 - The Unified Modeling Language. Lecture Notes in Computer Science, vol. 2863, pp. 2–17. Springer-Verlag (Oct 2003)
2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. Model Driven Engineering Languages and Systems pp. 436–450 (2007)
3. Feferman, S.: And so on... reasoning with infinite diagrams. Synthese (2010), to appear.
4. Goguen, J.: An Introduction to Algebraic Semiotics, with Application to User Interface Design, Lecture Notes in Computer Science, vol. 1562. Springer Berlin Heidelberg, Berlin, Heidelberg (Mar 1999)
5. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology 11(2), 256–290 (2002)
6. Lancaric, J., Ostroff, J., Paige, R.: The BON CASE tool. Details available via http://www.cs.yorku.ca/∼eiffel/bon_case_tool/ (Mar 2002)

7. Nerson, J.M.: Applying object-oriented analysis and design. Communications of the ACM 35(9), 63–74 (Sep 1992)
8. Owre, S., Rushby, J., , Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) 11th International Conference on Automated Deduction (CADE). Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer-Verlag, Saratoga, NY (Jun 1992)
9. Paige, R., Kaminskaya, L., Ostroff, J., Lancaric, J.: BON-CASE: an extensible CASE tool for formal specification and reasoning. Journal of Object Technology 1(3), 77–96 (2002)
10. Paige, R.F., Ostroff, J.: Metamodelling and conformance checking with PVS. In: Proceedings of Fundamental Aspects of Software Engineering. Lecture Notes in Computer Science, vol. 2029. Springer-Verlag (Apr 2001)
11. Skinner, R.: An Integrated Development Environment for the Business Object Notation (2010), MSc Thesis, University College Dublin
12. Snook, C., Butler, M.: UML-B: Formal modelling and design aided by UML. ACM Transactions on Software Engineering and Methodology 15(1), 92–122 (January 2006)
13. The Eclipse Modeling Framework Project: `http://http://www.eclipse.org/modeling/emf/` (2010)
14. Waldén, K., Nerson, J.M.: Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems. Prentice–Hall, Inc. (1995)