Constructing Models with the Human-Usable Textual Notation

Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack

Department of Computer Science, University of York, UK
{louis,paige,dkolovos,fiona}@cs.york.ac.uk

Abstract. We present an implementation of the OMG's Human-Usable Textual Notation (HUTN) [6] that provides a generic concrete syntax for MOF-based metamodels. The notation is summarised. Ways in which HUTN can be applied in order to improve the productivity of Model-Driven Engineering are identified. The use of HUTN to improve the quality of test suites for verifying model management operations (such as model-to-model transformation) is described. We also present a comparison of generic and domain-specific concrete syntax with HUTN.

1 Introduction

Whilst metamodel developers concentrate on the design of abstract syntax, the instantiation of a metamodel requires a concrete syntax. Concrete syntaxes can be classified as either *generic* or *domain-specific*. The former are defined at the level of metamodelling languages (e.g. MOF), while the latter are defined for specific metamodels. If a generic concrete syntax can be employed, then tool support can be provided for instantiating any metamodel, incorporating common features such as syntax highlighting and content assistance at no extra cost.

OMG's Human-Usable Textual Notation (HUTN) [6] defines such a generic concrete syntax, which aims to conform to human-usability criteria. However, we can find no current reference implementation of HUTN: the Distributed Systems Technology Centre's TokTok project (an implementation of the HUTN specification) seems to be inactive (the source code can no longer be found), whilst work on implementing the HUTN specification by Muller and Hassenforder [20] has been abandoned in favour of Sintaks [7], which operates upon domain-specific concrete syntax. Instead, metamodel developers use generation tools to derive specific syntax parsers and editors. Among these approaches, Eclipse technologies such as EMF [3] (and GMF [5]) can be used to generate tree-based (and graphical) model editors. Baar [1] introduces a formalisation of visual concrete syntax, and highlights its usefulness in analysis of correctness[1]. Text-to-model transformation tools such as xText [21], TCS [8] and TEF [25] can be used to define a custom concrete syntax for a metamodel and to generate a corresponding parser.

[1] For Baar, correctness means that each diagram (an instance of the visual concrete syntax) must correspond to only one model.

HUTN can be used when a domain-specific concrete syntax is inappropriate or unnecessary. For example, if a metamodel is developed incrementally, use of a generic concrete syntax avoids the need for frequent revision that would apply if a domain-specific concrete syntax were used. Such revisions can significantly detract from the productivity of incremental development, as the domain-specific concrete syntax must be adapted each time the abstract syntax changes. A generic concrete syntax such as HUTN is also appropriate for iterative test-driven development (or rapid prototyping) of models.

We present an overview of our new implementation of the HUTN specification, developed atop the existing Epsilon model management platform. We discuss the ways in which we are employing the HUTN implementation for prototyping, for comparing models, and in testing – all important tasks in Model-Driven Engineering (MDE). First, we summarise HUTN.

2 Human-Usable Textual Notation

HUTN defines a generic concrete syntax for constructing instances of MOF-based metamodels. In this section, we introduce the core syntax and the key features of HUTN. The complete definition is available at [6]. To illustrate usage of the notation, we use the MOF-based metamodel of families in Figure 1. (A nuclear family "consists only of a father, a mother, and children." [19])

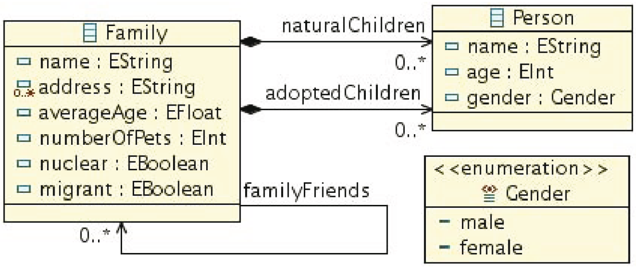

Fig. 1. Example metamodel: families and their children

2.1 Basic Notation

Listing 1.1 shows the construction of an *object* in HUTN, here an instance of the Family class from Figure 1. Line 1 specifies the package containing the classes to be constructed (`FamilyPackage`) and a corresponding identifier (`families`), used for fully qualifying references to objects (Section 2.3). Line 2 names the class (`Family`) and gives an identifier for the object (`The Smiths`). Lines 3 to 7 define *attribute values*; in each case, the data value is assigned to the attribute with the specified name. The encoding of the value depends on its type: strings are delimited by any form of quotation mark; multi-valued attributes use comma separators, etc.

The metamodel in Figure 1 defines a *simple reference* (familyFriends) and two *containment references* (adoptedChildren; naturalChildren). The HUTN representation embeds a contained object directly in the parent object, as shown in Listing 1.2. A simple reference can be specified using the type and identifier of the referred object, as shown in Listing 1.3. Like attribute values, both styles of reference are preceded by the name of the meta-feature.

```
FamilyPackage "families" {
  Family "The Smiths" {
    nuclear: true
    name: "The Smiths"
    averageAge: 25.7
    numberOfPets: 2
    address: "120 Main Street", "37 University Road"
  }
}
```

Listing 1.1. Specifying attributes with HUTN

```
FamilyPackage "families" {
  Family "The Smiths" {
    naturalChildren: Person "John" { name: "John" },
                     Person "Jo" { gender: female }
  }
}
```

Listing 1.2. Instantiation of naturalChildren – a HUTN containment reference

```
FamilyPackage "families" {
  Family "The Smiths" {
    familyFriends: Family "The Does"
  }
  Family "The Does" {}
}
```

Listing 1.3. Specifying a simple reference with HUTN

2.2 Keywords and Adjectives

While HUTN is unlikely to be as concise as a domain-specific concrete syntax, the notation does define a number of syntactic shortcuts in order to make model specifications more compact. These shortcuts can be used in place of more verbose (and more readable) full syntax. Shortcut use is optional, and their syntax is often intuitive [6, pg2-4]. Two example notational shortcuts are described here, in order to illustrate some of the ways in which HUTN can be used to construct models in a concise manner – a key concern when using HUTN to construct models for use in testing, as we will discuss in Section 3.

When specifying a *Boolean-valued attribute*, it is sufficient to simply use the attribute name (value `true`), or the attribute name prefixed with a tilde (value `false`). When used in the body of the object, this style of Boolean-valued attribute represents a *keyword*. A keyword used to prefix an object declaration is called an *adjective*. Listing 1.4 shows the use of both an attribute keyword (`~nuclear` on line 6) and adjective (`~migrant` on line 2).

```
1  FamilyPackage "families" {
2     nuclear ~migrant Family "The Smiths" {}
3
4     Family "The Does" {
5        averageAge: 20.1
6        ~nuclear
7        name: "The Does"
8     }
9  }
```

Listing 1.4. Using keywords and adjectives in HUTN

2.3 Inter-package References

To conclude our summary of the notation, we present two advanced features defined in the HUTN specification, which we have found to be useful when constructing large models. The first enables objects to refer to other objects in a different package, while the second provides means for specifying the values of a reference for all objects in a single construct (which can be used to improve the understandability of complex relationships in some cases). We conclude by discussing the way in which HUTN permits document customisation.

```
FamilyPackage "families" {
   Family "The Smiths" {}
}
VehiclePackage "vehicles" {
   Vehicle "The Smiths' Car" {
      owner: FamilyPackage.Family "families"."The Smiths"
   }
}
```

Listing 1.5. Referencing objects in other packages with HUTN

To reference objects between separate package instances in the same document, the package identifier is used in order to construct a fully qualified name. Suppose we introduce a second package to our metamodel in Figure 1. Among other concepts, this package introduces a Vehicle class, which defines an owner reference of type Family. Listing 1.5 illustrates the way in which the owner feature can be populated. Note that the fully qualified form of the class uses the names of elements of the metamodel, while the fully-qualified form of the object uses only HUTN identifiers defined in the current document.

The HUTN specification defines name scope optimisation rules, which allow the definition above to be simplified to: `owner: Family "The Smiths"`, assuming that (1) the VehiclePackage does not define a Family class, and (2) the identifier “The Smiths” is not used in the VehiclePackage block, or this HUTN document is configured to require unique identifiers over the entire document.

2.4 Alternative Reference Syntax

In addition to the syntax defined in Listings 1.2 and 1.3, the value of references may be specified independently of the object definitions. For example, Listing 1.6 demonstrates this alternate syntax by defining The Does as friends with both The Smiths and The Bloggs.

```
FamilyPackage "families" {
   Family "The Smiths" {}
   Family "The Does" {}
   Family "The Bloggs" {}

   familyFriends {
      "The Does" "The Smiths"
      "The Does" "The Bloggs"
   }
}
```

Listing 1.6. Using a reference block in HUTN

Listing 1.7 illustrates a further alternative syntax for references, which employs an infix notation.

```
FamilyPackage "families" {
   Family "The Smiths" {}
   Family "The Does" {}
   Family "The Bloggs" {}

   Family "The Smiths" familyFriends Family "The Does"
   Family "The Smiths" familyFriends Family "The Bloggs"
}
```

Listing 1.7. Using an infix reference in HUTN

2.5 Customisation Via Configuration

Some limited customisation of HUTN for particular metamodels can be achieved using *configuration files*. Customisations permitted include a parametric form of object instantiation (not yet implemented); renaming of metamodel elements; giving default values for attributes; and stating an attribute whose values are used to infer a default identifier. The HUTN specification [6] gives details of shortcuts and of the rules supported by configuration files.

In the next section, we motivate the need for a HUTN implementation in our development and testing of model management tools, reflecting on the shortcomings of other approaches, and deriving requirements for model specification.

3 Motivation: The Case of Model Construction for Testing

Generic concrete syntax can give benefit in a range of work with metamodels and model management, such as:

– rapid model production, used in investigating the appropriateness of metamodel design;
– visualisation of model comparisons; and
– provision of a model construction notation for testing model management tasks: e.g. model-to-model (M2M) or model-to-text (M2T) transformation.

Here, we focus on the last, the construction of models for testing operations of model management tools. We outline two non-HUTN approaches that we have used, and, in reflecting on their problems, motivate the use of HUTN, finally illustrating the sort of benefit that HUTN has brought to this work.

Model management is the discipline of managing assets involved in MDE. Model management tools both consume and produce models – for instance, a M2M transformation both produces and consumes models. Model management tools, like any software, are subject to rigorous testing. The test suite for a tool that consumes models requires sample models on which to run tests, whilst for a tool that produces models, models are needed against which to judge the test result (for a description of such verification, on model-to-model transformations, see [12,18]). Like any incremental and iterative development, changes in the tool being developed often requires change to the test suites. Equally, if a metamodel is updated, the models used by test suites require migration.

Before working with HUTN, we tried specifying test-suite models with EMF and in XMI. This caused problems when metamodels changed such that specified instances became invalid, as EMF's editor could not be used to migrate the specified models. Instead, the model migration involved hand-crafting changes in XMI.

A more successful approach (though, as we will see, not as flexible as HUTN) was to construct models programmatically. For instance, the code in Listing 1.8 constructs an instance of the Families metamodel, Figure 1, which has one family with two children.

```
@BeforeClass public static void constructModel() {
   final FamilyPackageFactory factory =
         FamilyPackageFactory.eINSTANCE;

   final Family family = factory.createFamily();
   family.setName("The Smiths");
```

```

    final Person john = factory.createPerson();
    john.setName("John");
    john.setGender(Gender.MALE);

    final Person jo = factory.createPerson();
    jo.setName("Jo");
    jo.setGender(Gender.FEMALE);

    family.getNaturalChildren().add(john);
    family.getNaturalChildren().add(jo);
}
```

Listing 1.8. A program to generate a model family using JUnit (Java)

Inclusion of the construction code for models as part of the test case reduces the number of files that needed to be edited when a test has to be updated. This simplifies the task of updating test cases. Also, in the programmatic approach, when changes to a metamodel render a model invalid, this is flagged by a compilation error on the test case. However, the inclusion of model construction code in the test case detracts from overall readability and conciseness.

In addition to wanting to improve the agility and responsiveness to change of our tool development we needed a way to specify model construction that was also also understandable, in order to improve maintainability. Therefore, the new approach to model specification needed to be:

– **concise**: to minimise the amount of code needed to construct a model;
– **readable**: by developers, so that they can easily understand test cases;
– **embeddable**: to support insular test cases (in which all the data needed to understand the test case is held in a single file), it must be possible to embed the model specification in the specification of a test case.

We now use HUTN for specification of test suite models: this provides human-usable, compact model code. For example, Listing 1.9 is the HUTN code for constructing the same model as that expressed programmatically in Listing 1.8.

```
FamilyPackage "families" {
   Family "The Smiths" {
      naturalChildren: male Person "John", female Person "Jo"
   }
}
```

Listing 1.9. Specifying a model using HUTN

As well as the obvious removal of Java code structures, the HUTN specification introduces helpful terminology to the test model, such as the `male` and `female` adjectives that specify values for the Gender type. Configuration rules were used to further reduce the size of the HUTN specification: the `name` attribute of Family and Person objects is assigned the value of the object's identifier.

Having demonstrated how HUTN supports our test-suite modelling requirements, we now turn to the HUTN implementation, which is essential for the realisation of non-manual test-suite model maintenance.

4 HUTN Implementation

Our implementation of HUTN makes extensive use of our Epsilon model management platform. We first summarise Epsilon and its component model management languages, before outlining the HUTN implementation itself.

4.1 The Epsilon Platform

Epsilon, the *E*xtensible *P*latform for *S*pecification of *I*ntegrated *L*anguages for m*O*del ma*N*agement [10], is a suite of tools for MDE. Epsilon comprises a number of integrated model management languages, supported by a common infrastructure, for performing tasks such as model merging, model transformation and inter-model consistency checking [11]. Whilst many model management languages are bound to a particular subset of modelling technologies [9], Epsilon is technology-agnostic – models written in any modelling language can be manipulated by Epsilon. Models implemented using EMF, MOF 1.4, pure XML, or CZT [23] are currently supported.

The core of the platform is the Epsilon Object Language (EOL) [14], a reworking and extension of OCL that includes the ability to update models, conditional and loop statements, statement sequencing, and access to standard output and error streams. The Epsilon task-specific languages are built atop EOL, giving highly efficient inheritance and reuse of features. Currently, these task-specific languages include support for model-to-model transformation (ETL [16]), model comparison (ECL), model merging (EML [13]) model-to-text transformation (EGL [24]) and model validation (EVL [15]).

4.2 Implementation of HUTN Using Epsilon

Our implementation of HUTN uses the task-specific languages of Epsilon. Although any languages for model-to-model transformation (M2M), model-to-text transformation (M2T) and model validation could have been used, Epsilon's existing domain-specific languages make the implementation a relatively simple exercise, and has the added advantage that the HUTN tool is directly compatible with the Epsilon model management facilities. Our implementation has been released as part of Epsilon, and includes development tools for Eclipse (a source code editor and builder).

Figure 2 outlines the workflow through our HUTN implementation, from HUTN source text to instantiated target model. The HUTN model specification is parsed to an abstract syntax tree using a HUTN parser specified in ANTLR [22]. From this, a Java postprocessor is used to construct an instance of a simple AST metamodel (which comprises two meta-classes, Tree and Node). Using

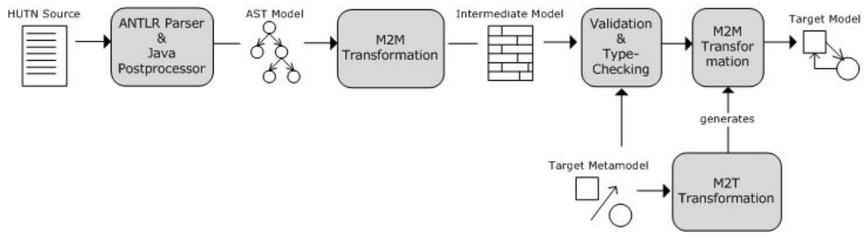

Fig. 2. The architecture of our HUTN implementation

ETL, M2M transformations are then applied to produce an intermediate representation from the AST model. Finally, a M2T transformation on the target metamodel, specified in EGL, produces a further M2M transformation, from the intermediate representation to the target model.

The workflow includes two M2M transformation stages, because the form of the AST metamodel is not suited to a one-step transformation. There is a mismatch between the features of the AST metamodel and the needs of the target model – for example, between the Node class in the AST metamodel and classes in the target metamodel. If a one-step transformation was used, each transformation rule would need a lengthy guard statement, which is hard to understand (and to verify). Instead, we designed a fine-grained representation of the HUTN AST as an intermediate: the instance of the AST metamodel is first transformed to this fine-grained representation (the *intermediate model*), and that representation is transformed in to the target model.

We now discuss the two M2M transformations in depth, along with a model validation phase which is performed prior to the second transformation.

AST Model to Intermediate Model. In the HUTN implementation, M2M transformation rules are written in ETL. An example transformation rule is shown in Listing 1.10. The rule transforms a name node in the AST model (which could represent a package or a class object) to a package object in the intermediate model. The guard (line 5) specifies that a name node will only be transformed to a package object if the node has no parent (i.e. it is a top-level node, and hence a package rather than a class). The body of the rule states that the type, line number and column number of the package are determined from the text, line and column attributes of the node object. On line 11, a containment slot is instantiated to hold the children of this package object. The children of the node object are transformed to the intermediate model (using a built-in method, `equivalent()`), and added to the containment slot.

```
1  rule NameNode2PackageObject
2     transform n : AntlrAst!NameNode
3     to p : Intermediate!PackageObject {
4
5     guard : n.parent.isUndefined()
6
```

```
    p.type := n.text;
    p.line := n.line;
    p.col := n.column;

    var slot := new Intermediate!ContainmentSlot;
    for (child in n.children) {
       slot.objects.add(child.equivalent());
    }
    if (slot.objects.notEmpty()) {
       p.slots.add(slot);
    }
}
```

Listing 1.10. Transformation rule (in ETL) to convert AST nodes to package objects

Intermediate Model Validation. An advantage of the two-stage transformation is that contextual analysis can be specified in an abstract manner – that is, without having to express the traversal of the AST. This gives clarity and minimises the amount of code required to define of our constraints.

```
context ClassObject {
   constraint IdentifiersMustBeUnique {
      guard: self.id.isDefined()
      check: ClassObject.allInstances()
                .select(c|c.id = self.id).size() = 1;
      message: 'Duplicate identifier: ' + self.id
   }
}
```

Listing 1.11. A constraint (in EVL) to check that all identifiers are unique

We use EVL [15] to write verification statements to perform this analysis in our implementation of the HUTN specification, resulting in highly expressive syntactic constraints. An EVL constraint comprises a guard, the logic that specifies the constraint, and a message to be displayed if the constraint is not met. For example, Listing 1.11 specifies the constraint that every HUTN class has a unique identifier.

Intermediate Model to Target Model. Because the contextual analysis is performed on the intermediate model, we have assurance that this is a valid model before proceeding to the target model. In generating the target model from the intermediate model (Figure 2), the transformation uses information from the target metamodel, such as the names of classes and features. A typical approach to this category of problem is to use a higher-order transformation on the target metamodel to generate the desired transformation. In this case, we use a different approach, which we consider to be more readable: the transformation to the target model is produced by executing an EGL template on the target metamodel. EGL is a template-based text generation language. `[% %]` tag pairs are used to denote dynamic sections, which may produce text when executed. Any code not enclosed in a `[% %]` tag pair is included verbatim in the generated text.

Listing 1.12 is the EGL template for a M2T transformation on the target metamodel; it generates the M2M transformation used for generating the target model. The loop beginning on line 1 iterates over each meta-class in the metamodel, producing a transformation rule to generate target model instances of that meta-class from class objects in the intermediate model. The template guard (line 6) specifies that only class objects of the same type as the meta-class be transformed by the current rule. The template body is omitted from Listing 1.12 – it iterates over each structural feature of the current meta-class, and generates appropriate transformation code for populating the values of each structural feature from the slots on the class object in the intermediate model.

```
 1  [% for (class in EClass.allInstances()) { %]
 2  rule Object2[%=class.name%]
 3    transform o : Intermediate!ClassObject
 4    to t : Model![%=class.name%] {
 5
 6      guard: o.type = '[%=class.name%]'
 7
 8      -- body omitted for brevity
 9    }
10  [% } %]
```

Listing 1.12. Initial sections of the template (in EGL) for generating rules (in ETL) to instantiate classes of the target metamodel

4.3 Discussion of HUTN Implementation

Like Muller and Hassenforder [20], we encountered a number of problems with the HUTN specification. Where we have encountered the same problem, we have employed their solutions.

A further problem encountered relates to the syntactic shortcut for HUTN adjectives. The HUTN [6] specification includes the grammar rules shown in Listing 1.13. The `AttributeName` and `ClassName` terminals produce the same token type. Therefore, the adjectival shortcut for a Boolean valued attribute with the value `true` is syntactically the same as the name of a class. For example, when parsing the expression, `nuclear`, it is not possible to determine whether to match an `AttributeName` or a `ClassName`.

```
ClassHeader ::= ClassAdjectives <ClassName> (ClassIdentifier)?
ClassAdjectives ::= (('~'? <AttributeName>) | DataValue)*
```

Listing 1.13. Extract from the HUTN grammar (taken from [6])

The solution that we adopt is to modify the HUTN grammar to require that such adjectives be prefixed with a colon. Thus, for example, the expression, `nuclear ~migrant Family "The Smiths" {}`, is rewritten as,
`#nuclear ~migrant Family "The Smiths" {}`.

5 Discussion and Related Work

As noted previously, there are many tools that aid developers in constructing instances of a metamodel, including xText [21], TCS [8] and TEF [25]. Typically, these tools are used for model-specific concrete syntax; the initial overhead in generating editors and parsers, and designing the concrete syntax, is significant, and there is a further cost every time the metamodel or syntax is modified.

Some generated editors provide other model management facilities. For instance, oAW's text-to-model tool, xText [21] includes the Check language, which, in its approach to specifying domain-specific constraints on concrete syntax, is similar to our HUTN implementation's use of EVL (see Listing 1.11).

The Eclipse Modelling Framework (EMF) [3] provides an alternative to generated editors. EMF provides a reflective, tree-structure editor that can be used to instantiate a metamodel in a dynamic manner (the objects used to construct the model are not strongly-typed).

However, EMF's reflective editor does not provide any means for customisation. By contrast, Epsilon provides an enhanced reflective editor, Exeed, that permits such customisation via metamodel annotations. Our HUTN tooling provides an alternative reflective editor – one that recognises a textual, rather than a tree-structured, concrete syntax.

Epsilon's EOL can be used, with EUnit [17] – a task-specific language for testing built atop Epsilon – to perform unit testing of model management operations. To date, the models used in conjunction with eUnit have been specified using EMF editors. We have added a driver to the Epsilon Model Connectivity layer that connects EUnit and our HUTN implementation, so that Epsilon can support manipulation of models authored in HUTN. The driver allows Epsilon users to use eUnit to specify assertions in test cases and HUTN to construct models to be used in test suites alongside these assertions.

In relation to our use of HUTN in test suite development, Schuh and Punke [26] present an object-oriented programming pattern that describes a similar style of test model creation, called the ObjectMother pattern. They report that the standardisation of object creation improved the maintainability of test suites. They also note that by allowing tests to slightly change a fixture without defining a completely new object (e,g. use the Family, The Smiths, without the Person John as a child) it was possible to further improve the quality of test suites.

HUTN provides a generic concrete syntax for metamodels that conform to MOF. A generic concrete syntax provides a large degree of flexibility, and can be used as a notation for a number of distinct abstract syntax. It is useful where a domain-specific concrete syntax is inappropriate or unnecessary.

Notwithstanding the power of genericity, there are situations where a domain-specific concrete syntax is preferable. An example of where HUTN is unhelpful arose when developing a metamodel for the recording of failure behaviour of components in complex systems, based on the work of Wallace [27].

Failure behaviours comprise a number of expressions that specify how each component reacts to system faults, and there is an established concrete syntax for expressing failure behaviours. The failure syntax allows various shortcuts, such as the use of underscore to denote a wildcard. For example, the syntax for a possible failure behaviour of a component that receives input from two other components (on the left-hand side of the expression), and produces output for a single component is denoted:

$$(\{\_\}, \{\_\}) \rightarrow (\{late\}) \tag{1}$$

A failure behaviour can contain many expressions, and each component may be connected to many other components, so the metamodel for failure behaviours contains a large number of classes. In the generic concrete syntax, the specification of these behaviours is unhelpfully terse. For example. Listing 1.14 gives the HUTN syntax for failure behaviour (1), above.

```
Behaviour {
   lhs: Tuple {
      contents: IdentifierSet { contents: Wildcard {} },
                IdentifierSet { contents: Wildcard {} }
   }

   rhs: Tuple {
      contents: IdentifierSet { contents: Fault "late" {} }
   }
}
```

Listing 1.14. Failure behaviour specified in HUTN

In general, HUTN is less concise than a domain-specific syntax for metamodels containing a large number of classes with few attributes, and in cases where most attributes are used to define structural relationships among concepts. However, there might still be benefits from using HUTN in such cases, if the metamodel is likely to be modified frequently, or if the model does not yet have a formal metamodel.

6 Conclusion and Further Work

We have shown that OMG's HUTN is a valuable addition to the facilities available to metamodel developers, giving genericity and flexibility in modelling. We present our implementation of HUTN in the context of our primary motivation – of supporting models associated with test suites for tools associated with the Epsilon model management platform. We summarised the way in which our implementation has improved the quality of our test suites in MDE.

Future directions for our work with HUTN include improvements to our test-suite-model support, along the lines described by Schuh and Punke (above). We are also planning to aid a mechanism for specifying that an non-deterministic initialisation of features. Non-deterministic initialisation will permit features whose values are unimportant to the current test to be initialised automatically by the HUTN tooling.

We are also working on an extension of the HUTN implementation that supports *metamodel inference*. Currently, we have implemented the metamodel inference algorithm as a rudimentary transformation from the intermediate model to an Ecore [2] metamodel. We intend to use metamodel inference in prototyping model-centric systems.

Additionally, we are investigating other uses for HUTN, in particular: visualising the differences between versions of models: a HUTN view could be contributed to the EMF Compare platform [4]. We intend to use such visualisation to help identify categories of model and metamodel evolution.

Acknowledgement. The work in this paper was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the "Information Society Technologies" Sixth Framework Programme (2006-2009).

References

1. Baar, T.: Correctly Defined Concrete Syntax for Visual Modeling Languages. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 111–125. Springer, Heidelberg (2006)
2. IBM Corporation. Ecore API Documentation (2006), `http://download.eclipse.org/modeling/emf/emf/javadoc/2.4.0/org/eclipse/emf/ecore/package-summary.html`
3. The Eclipse Foundation. Eclipse Modelling Framework Project (2008), `http://www.eclipse.org/modeling/emf/`
4. The Eclipse Foundation. EMF Compare (2008), `http://wiki.eclipse.org/index.php/EMF_Compare`
5. The Eclipse Foundation. Graphical Modelling Framework (2008), `http://www.eclipse.org/modeling/gmf/`
6. Object Management Group. Human-Usable Textual Notation Specification (2004), `http://www.omg.org/technology/documents/formal/hutn.htm`
7. IRISA. Sintaks (2007), `http://www.kermeta.org/sintaks/`
8. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes. In: Proc. GPCE 2006, pp. 249–254. ACM Press, New York (2006)
9. Kolovos, D.S.: A Short Introduction to Epsilon (2007), `http://www.cs.york.ac.uk/~dkolovos/epsilon/Epsilon.ppt`
10. Kolovos, D.S.: Extensible Platform for Specification of Integrated Languages for mOdel maNagement Project Website (2007), `http://www.eclipse.org/gmt/epsilon`
11. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Epsilon Development Tools for Eclipse. In: Eclipse Summit 2006 (2006)
12. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model comparison: a foundation for model composition and model transformation testing. In: Proc. GaMMa 2006, pp. 13–20. ACM Press, New York (2006)
13. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Merging Models with the Epsilon Merging Language (EML). In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 215–229. Springer, Heidelberg (2006)

14. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
15. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: On the Evolution of OCL for Capturing Structural Constraints in Modelling Languages. In: Workshop on Rigorous Methods for Software Construction & Analysis. LNCS, vol. 5115. Springer, Heidelberg (2008)
16. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063. Springer, Heidelberg (2008)
17. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Unit Testing Model Management Operations. In: Proc. MoDeVVa, ICST. IEEE, Los Alamitos (2008)
18. Lin, Y., Zhang, J., Gray, J.: A Framework for Testing Model Transformations. In: Model-driven Software Development, pp. 219–236. Springer, Heidelberg (2005)
19. Merriam-Webster. Definition of Nuclear Family (2008), `http://www.merriam-webster.com/dictionary/nuclear%20family`
20. Muller, P.-A., Hassenforder, M.: HUTN as a Bridge between ModelWare and GrammarWare. In: WISME Workshop, MODELS / UML 2005 (2005)
21. OpenArchitectureWare. openArchitectureWare Project Website (2008), `http://www.eclipse.org/gmt/oaw/`
22. Terence Parr. ANTLR Parser Generator (2008), `http://www.antlr.org/`
23. Community Z Tools Project. Community Z Tools (2007), `http://czt.sourceforge.net/`
24. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The Epsilon Generation Language. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 1–16. Springer, Heidelberg (2008)
25. Scheidgen, M.: Textual Modelling Embedded into Graphical Modelling. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 153–168. Springer, Heidelberg (2008)
26. Schuh, P., Punke, S.: ObjectMother: Easing Test Object Creation in XP. In: XP Universe (2001)
27. Wallace, M.: Modular architectural representation and analysis of fault propagation and transformation. Electr. Notes Theor. Comput. Sci. 141(3), 53–71 (2005)