Automated formalisation for verification of diagrammatic models

James R. Williams [1,2]

Department of Computer Science
University of York, York, UK

Fiona A. C. Polack [3]

Department of Computer Science
University of York, York, UK

Abstract

Software engineering uses models to design and analyse systems. The current state-of-the-art, various forms of model-driven development, uses diagrams with defined abstract syntax but relatively-lose translational approaches to semantics, which makes it difficult to perform rigorous analysis and verification of models. Here, we present work-in-progress on tool support for formal verification of diagrammatic models. The work builds on Amálio's rigorous template-based approach to formalisation, which formally expresses the intended semantics of both the diagram notation and modelled system, along with standard correctness conjectures and, in many cases, proof of these conjectures.

Keywords: formal verification, model-driven development, tool-support

1 Introduction

In practical software engineering, diagrammatic approaches are widely used for sketching, specifying and designing systems. There are challenges in applying formal analysis to these approaches, either because the semantics is inadequately defined, or because the level of detail of the semantics does not admit interesting or useful formal analysis. Conversely, the formal notations used in verification of critical systems and in academia are considered inaccessible by many practical engineers [21,12,27]. If it were possible to provide access to formal analysis without great

[1] We would like to thank: Dimitrios Kolovos, without whose help and Epsilon tool-suite we could not have produced this tool; Nuno Amálio, Faculté des Sciences, de la Technologie et de la Communication, University of Luxembourg, whose doctoral work on GeFoRME and FTL form the basis of the tool; Leo Freitas and Ian Toyn, for advice on CZT and CADiZ; Richard Paige for comments on earlier drafts; and the UK National Research and Training Initiative: Large-Scale Complex IT Systems for part sponsorship.

[2] Email: james@jamesrobertwilliams.co.uk

[3] Email: fiona@cs.york.ac.uk

cost, the rigour and reliability of diagrammatic models could be significantly enhanced. For many years, people have sought to combine formal and diagrammatic approaches so that formal type-checking and proof of properties can be used to explore the correctness and internal consistency of diagrams. However, few integrations have found favour, either because formalisation does not meet engineering needs, or because engineers are exposed to the formalisation and formal model.

In an attempt to bring formalisms into practical engineering, Amálio [3] proposes a generative framework for rigorous model engineering (GeFoRME) that applies template-based translation to diagrammatic models. Formal templates capture the intended semantics of diagram concepts as well as the modelled system. A well-founded template language (the formal template language, FTL [3,8]) makes it possible to generate not only formal models but also standard consistency conjectures and, in many cases, their proofs. Amálio demonstrates an instance of GeFoRME called UML+Z, comprising a library of templates that capture the object semantics of UML. Instantiating the templates with details from UML diagrams results in a Z model that conforms to Amálio's object-oriented (OO) Z structuring, ZOO [4,7,3,10]. UML+Z has been used on small case studies [3,9] of conventional object-oriented models including UML class and state diagrams and Catalysis [19] snapshots. It has also been used in an attempt to formalise part of the meta-object facility [2] and on a model of autonomous objects [10]. The GeFoRME approach provides ease of construction, strong semantic support, and traceability between the formal and diagrammatic models. If templates adequately capture the semantics of source-model concepts, and the source model is syntactically correct, then the transformation produces a type-correct formal model, on which the conjectures constructed by template translation are true by construction [6,8].

In all the work with UML+Z, however, a major inhibiting factor is the lack of tool support for management and instantiation of templates. The GeFoRME approach employs a formal *translation*. Translation is a conventional way of considering "integrated methods" (reviewed in, for example, [17]), but it is essentially the same as the *model transformations* that are characteristic of model-driven development (MDD) – the family of approaches to tool-supported practical software development that focuses on the construction and manipulation of primarily-diagrammatic models [29]. As in formal translation, MDD model transformation defines a mapping from a source model to a target model [11]. The mapping, or transformation definition, comprises transformation rules which specify how each modelling construct is transformed. In transforming diagrammatic models to formal models, we apply a model-to-text (as opposed to model-to-model) transformation. Model-to-text transformation is most commonly used for code generation [1], so is appropriate for generating, for instance, the LaTeX (or any other) markup for a formal notation.

This paper presents a tool for the management and instantiation of FTL templates. The tool currently supports the UML+Z translation, as well as a means to extend the template base; the tool and GeFoRME are sufficiently generic that the approach could also support other diagrammatic source and formal target languages. The tool provides the interaction necessary for a naïve user to generate a formal model and automatically verify it using existing Z tools. In addition, the tool supports expert users who wish to extend the template repertoire or provide new specialisations of the tool.

In the next section, we introduce Amálio's template-based formalisation. In section 3 we describe the tool support, and in section 4 we illustrate the capabilities of UML+Z, the translation tool and support for different user levels. In section 5, we discuss the advantages and limitation of the work so far, and consider the extensions to the tool and the template library.

2 Amálio's Template-based Formalisation

In [9], Amálio motivates his work by identifying some shortcomings of diagrammatic approaches to software engineering that can be addressed by practical formalisation. He notes the need to model concepts that cannot be expressed diagrammatically – constraints and model properties – and considers the general problem of semantics. Diagrammatic approaches such as entity-relationship modelling and UML are widely used in practical software engineering, and the notations have well-defined abstract syntax (for instance, UML is defined using metamodels, based on the concept of reflectivity). However, the semantics of concepts such as *class*, *association* and *generalisation*, are under-specified. In most cases, the semantics is clarified only when a diagrammatic model is converted to code – thus, a class diagram that forms the basis for a Java program assumes a Java semantics, but the same diagram used to create a C++ program assumes a C++ semantics. In the case of UML (and related domain-specific modelling notations), annotations are added to diagrams using notations such as the object constraint language, OCL. The relationship between the modelling notations and the constraint language is defined at the abstract syntax level. An obvious connotation of this situation is that whilst properties of models can be demonstrated informally (for example, by animation), analysis of consistency or model properties is not consistent across modellers or modelling tools. In practice, little attention is paid to consistency across model-views or between diagrams and constraint expressions. Note that this problem is not unique to diagrammatic models: insufficient semantic underpinning was also a problem for Z, ultimately addressed in its ISO standardisation.

Most attempts to associate formal semantics to diagrammatic modelling assign a specific formal meaning to each diagrammatic concept. Whilst this significantly reduces ambiguity, and admits formal analysis, the formalisation assumes a single, fixed semantics for each concept [9] – and the semantics that is assumed is often only apparent to the formalist. One well-known example is the UML to Object-Z translation, which imposes the semantics of Object-Z inheritance on UML generalisation [15,14]. By contrast, Amálio's approach [3] builds on ideas of pattern-based development [20] and problem-driven methods [22] to advocate a framework for rigorous, but practical MDD [9]; concept semantics are captured explicitly in the templates, so a different semantics simply requires use of a different set of templates. The traceability provided by the template transformation approach allows engineers to work with the diagrams, which, in effect, form a graphical interface for the formality that lies beneath. In many cases, Amálio's approach allows the formalism to be completely hidden from the developer.

Amálio devised the Formal Template Language (FTL) [3,8] as the rigorous underpinning to the GeFoRME framework, supporting proof with template representations. Having captured patterns of formal development (e.g. a model structure) in FTL, reasoning can be applied at the pattern level using meta-proof. For example, a precondition of an operation can be calculated or an initialisation conjecture proved to establish meta-theorems that apply when the concept templates are instantiated [3,8]. The FTL templates and translation process are illustrated in Figure 1 (which also shows the subsequent automation, covered in section 3).

The existing UML+Z templates and meta-theorems relate to Amálio's ZOO structuring for Z [7,3]. ZOO uses standard Z – and can easily adapt to pre-standardisation dialects such as the Z/Eves variant. Unlike Object-Z [30], ZOO does not require any extension to the Z language or its tool support. Amálio shows how a ZOO model is built incrementally using template instantiation. Structural components are views representing the main OO concepts: objects, classes, associations and system. An object is an atom, a member of the set of all possible objects and of the set of possible objects of its class [10]. The class structure uses a *promoted Z abstract data type* [16]; it is represented by an intensional structure that defines the common properties of the class's objects, and an extensional structure, that defines the class as the set of its objects. The formal association structure forms tuples of the classes' objects using a Z relation. A system is an ensemble of classes and associations. Additional properties (constraints, conjectures) are expressed on the appropriate views.

2.1 Template Instantiation: Amálio's Bank Case Study

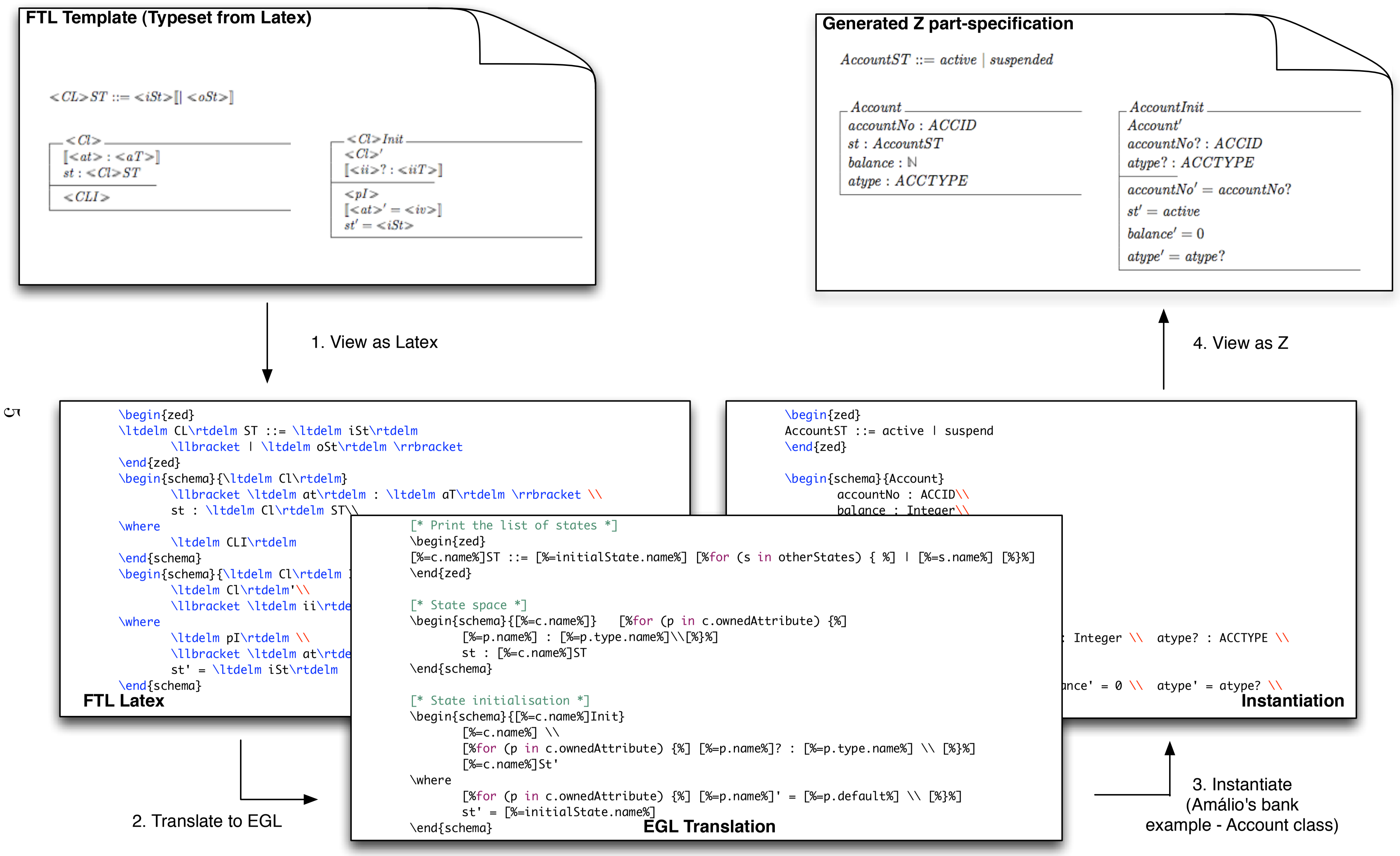
One of Amálio's case studies is a banking system. Amálio's formalisation of two classes from his example demonstrates in outline how template instantiation works. Amálio gives further descriptions in [3,9,8,6,4].

The class diagram, figure 2, shows two classes – *Customer* and *Account*. The association, *holds*, allows an instance of *Customer* to have zero or more accounts, and an instance of *Account* to have exactly one customer. The *Account* class has operations to *withdraw* money, to *deposit* money, to *getBalance* for an account; to *suspend* and to *reactivate* an account. The state changes caused by the *Account* operations are described in figure 3. The formalisation is demonstrated here for the *Account* class, illustrated in figure 1.

Select FTL templates The templates needed to instantiate the *Account* class comprise those for the intensional and extensional definition of a class, and those for the initialisation of the class (initialisation is a formal technique that captures a potential start-state for the system, and then proves that this is a valid state of the system) – see "FTL Template (Typeset from Latex)" in figure 1.

Select class diagram concepts and instantiate templates Each template is instantiated by replacing placeholders (e.g. $\lessdot x \gtrdot : \lessdot t \gtrdot$) with relevant concept and type names from the UML diagram. Where a template includes a FTL list (e.g. $[\![\ldots]\!]_{(sep,empty)}$), the statement is instantiated once for each element – shown in figure 1 as the instantiation of attributes of the *Account* class.

Select operation templates and identify operation effects For each operation on a class, the template instantiation depends on the type of operation.


FTL Template (Typeset from Latex)
Generated Z part-specification
1. View as Latex
4. View as Z
FTL Latex
Instantiation
EGL Translation
2. Translate to EGL
3. Instantiate (Amálio's bank example - Account class)

$<CL>ST ::= <iSt> [| <oSt>]$

```
<Cl>
[<at> : <aT>]
st : <Cl>ST
----
<CLI>

<Cl>Init
<Cl>'
[<ii>? : <iiT>]
----
<pI>
[<at>' = <iv>]
st' = <iSt>
```

$AccountST ::= active \mid suspended$

```
Account
accountNo : ACCID
st : AccountST
balance : ℕ
atype : ACCTYPE

AccountInit
Account'
accountNo? : ACCID
atype? : ACCTYPE
----
accountNo' = accountNo?
st' = active
balance' = 0
atype' = atype?
```

```
\begin{zed}
\ltdelm CL\rtdelm ST ::= \ltdelm iSt\rtdelm
        \llbracket | \ltdelm oSt\rtdelm \rrbracket
\end{zed}
\begin{schema}{\ltdelm Cl\rtdelm}
        \llbracket \ltdelm at\rtdelm : \ltdelm aT\rtdelm \rrbracket \\
        st : \ltdelm Cl\rtdelm ST\\
\where
        \ltdelm CLI\rtdelm
\end{schema}
\begin{schema}{\ltdelm Cl\rtdelm
        \ltdelm Cl\rtdelm'\\
        \llbracket \ltdelm ii\rtde
\where
        \ltdelm pI\rtdelm \\
        \llbracket \ltdelm at\rtde
        st' = \ltdelm iSt\rtdelm
\end{schema}
```

```
\begin{zed}
AccountST ::= active | suspend
\end{zed}

\begin{schema}{Account}
        accountNo : ACCID\\
        balance : Integer\\
: Integer \\  atype? : ACCTYPE \\
nce' = 0 \\  atype' = atype? \\
```

```
[* Print the list of states *]
\begin{zed}
[%=c.name%]ST ::= [%=initialState.name%] [%for (s in otherStates) { %] | [%=s.name%] [%}%]
\end{zed}

[* State space *]
\begin{schema}{[%=c.name%]}    [%for (p in c.ownedAttribute) {%]
        [%=p.name%] : [%=p.type.name%]\\[%}%]
        st : [%=c.name%]ST
\end{schema}

[* State initialisation *]
\begin{schema}{[%=c.name%]Init}
        [%=c.name%] \\
        [%for (p in c.ownedAttribute) {%] [%=p.name%]? : [%=p.type.name%] \\ [%}%]
        [%=c.name%]St'
\where
        [%for (p in c.ownedAttribute) {%] [%=p.name%]' = [%=p.default%] \\ [%}%]
        st' = [%=initialState.name%]
\end{schema}
```

Fig. 1. The steps of a UML+Z Translation, with examples of the type-set and LaTeX formats involved

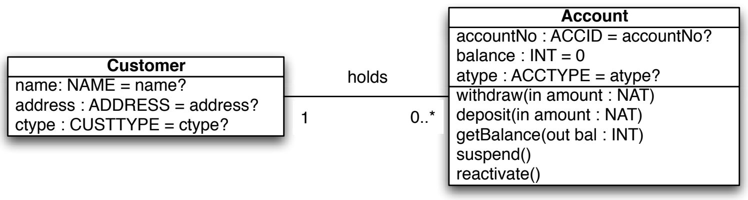

Customer
name: NAME = name?
address : ADDRESS = address?
ctype : CUSTTYPE = ctype?
holds
1
0..*
Account
accountNo : ACCID = accountNo?
balance : INT = 0
atype : ACCTYPE = atype?
withdraw(in amount : NAT)
deposit(in amount : NAT)
getBalance(out bal : INT)
suspend()
reactivate()

Fig. 2. Extract from Amálio's bank system class diagram [3]

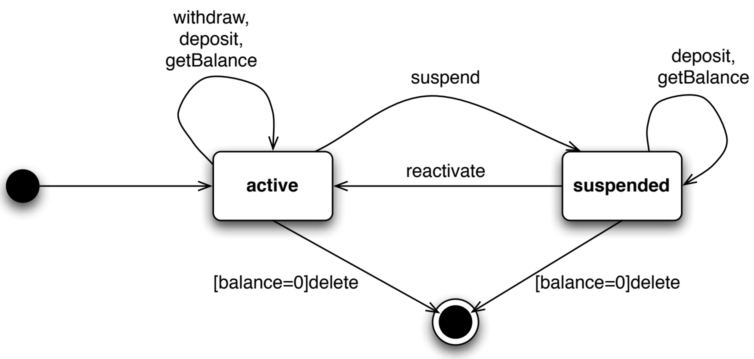


Fig. 3. Extract from Amálio's bank system state diagram [3]

Details of the operation are not included in class diagrams, but can be extracted from other UML diagrams or from diagram annotations, or can be entered by the user. For instance, the *withdraw* operation is identified as an `update` operation that changes the state of the system (the alternative is `observe`). Here, the attribute value that changes is *balance*, and the operation uses the formula, $balance - amount?$, where $amount?$ is the input to the operation.

Identify states from state diagram The state diagram shows which system states need to be represented, but these have to be interpreted in terms of the classes and attributes. For instance, in figure 3, the two states are *active* and *suspended*. The appropriate template instantiation adds an "attribute", *st: accountST*, to the formal representation of *Account*. Template instantiation then constructs the formal definition of the attribute type *accountST*, which here results in the expression, $accountST ::= active|suspended$, shown in figure 1.

Instantiate initialisation templates Initialisation in Z is an operation that has only an after-state, represented in post-condition predicates. Unlike other operations, it can be generated automatically by setting all attributes to a default value or an input. The initialisation template is shown in "FTL Template (Typeset from Latex)" of figure 1, and its instantiation is shown in the "Generated Z part-specification". Note that primed components (e.g. *Account'*) are the after-states of an operation, and queried components (e.g. *atype?*) are inputs. The initialisation of the state attribute *st'* is derived from the state diagram, since the start event in the state diagram points to the first state of the system – in the *Account* example, *st'* = *active*. In addition to the initialisation specification shown in figure 1, a Z conjecture is generated, that the initialisation state is a valid state of the system. This is associated to one of the meta-theorems (discussed above), and is true by construction if the Z model is type-correct.

3 The AUtoZ Tool

The AUtoZ tool (www.jamesrobertwilliams.co.uk/autoz.php) provides automation for Amálio's UML+Z process, outlined in section 2.1. This section describes design criteria for the tool. It then outlines the design of the *Automatic formalisation of UML to Z* tool framework, AUtoZ.

3.1 Tool Rationale

The motivation of enhancing the rigour of practical software engineering means that the tool has to be readily accessible to software engineers. The tool should integrate with existing software engineering tools, and should be able to inter-operate directly with existing diagramming and formal support tools.

Amálio's GeFoRME is a generic framework, which can be specialised for different diagrammatic and formal notations, so the tool needs to be modifiable to other notations.

The tool needs to catalogue the existing UML+Z templates efficiently and support the addition of new templates and meta-proofs: there is potential to extend the existing UML+Z templates with a wide range of alternative concept semantics. This requirement highlights the need to provide support for both basic transformation-and-analysis use, and expert maintenance of the template libraries.

3.2 Tool Design

The AUtoZ tool is a framework on which different instance tools can be built. The two instances that currently exist are AUtoCADiZ and AUtoZ/Eves. AUtoZ is a plug-in for the Eclipse development environment, which is used as the basis for many modelling and model-management activities in software engineering.

The tool requires serialised input of a diagram whose concepts (abstract syntax) can be selected. Common MDD diagramming tools provide serialised XMI output, and are underpinned by a metamodel that defines the abstract syntax (concepts) of the notation. This allows concepts and their labels to be automatically extracted from diagrams, to instantiate the FTL templates. Here, we use the existing Eclipse modelling plug-in, UML2 (www.eclipse.org/uml2/), since it supports UML 2.x modelling, uses a standard metamodel-based approach to abstract syntax, and sits within the Eclipse development environment. However, in principle AUtoZ could use XMI output from many other modelling tools.

To represent the FTL templates as model transformation mappings, we convert the FTL format to Epsilon EGL. Epsilon (www.eclipse.org/gmt/epsilon) provides a suite of integrated model-management tools [25,24], as part of the Eclipse-GMT project. EGL, the Epsilon Generation Language, supports model-to-text transformation. An EGL *run configuration* specifies a file containing the XMI source model, and a file containing EGL transformation rules to be executed on the source model, as described in [26]. The AUtoZ Eclipse plug-in is a customised EGL run configuration that executes the transformations on a UML2 XMI source model to

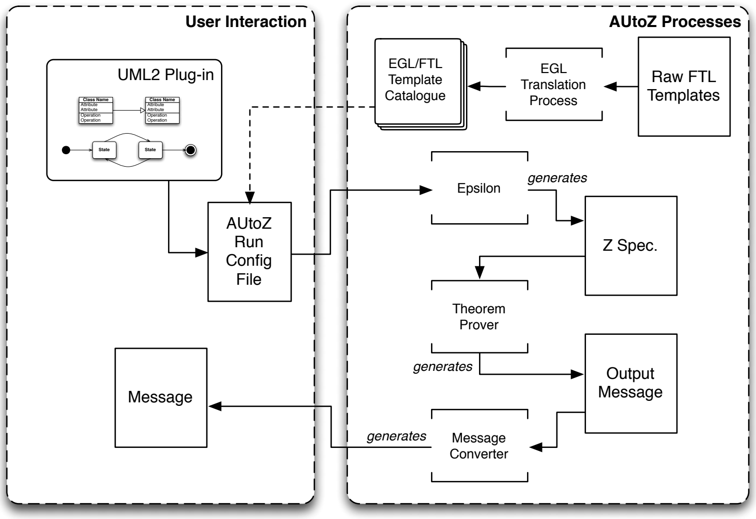


Fig. 4. AUtoZ workflow. UML2 and Epsilon are existing Eclipse facilities. The theorem prover can be any suitable formal analysis tool, and is called automatically by the AUtoZ tool instance.

Table 1
Some of the FTL-to-EGL conversions

FREE TEXT:	FTL (typeset)	$subCl : CLASS \to CLASS$
	FTL (LaTeX)	`subCl : CLASS \to CLASS`
	EGL	`subCl : CLASS \to CLASS`
PLACEHOLDER:	FTL (typeset)	$\ll x \gg : \ll t \gg$
	FTL (LaTeX)	`\ltdelm x \rtdelm : \ltdelm t \rtdelm`
	EGL	`[%=x.name%] : [%=x.type.name%]`
LIST:	FTL (typeset)	$\llbracket \ldots \rrbracket_{(sep,empty)}$
	FTL (LaTeX)	`\llbracket ...\rrbracket_{sep,empty}`
	EGL	`[% var append : Boolean := false;`
		`for (...){`
		`if (append){%]` *sep* `[%}`
		`...;`
		`append := true;`
		`} %]`

generate Z LaTeX markup, which is then input directly to the formal support tool for analysis. Figure 4 summarises the workflow of AUtoZ. A developer provides the UML model; the tool calls an AUtoZ run configuration file; Eclipse plug-ins convert UML to Z LaTeX via EGL, and call the relevant Z tool. The result of the formal analysis is returned to the user.

The EGL transformation rules are also templates, in a format that is very similar to FTL (though without any formal underpinning). Three typical FTL templates are shown in Table 1 with their LaTeX source, and the corresponding EGL rules. Many of the FTL concepts have a direct equivalent in EGL, and others, such as FTL lists, follow a common pattern in EGL.

The component architecture of AUtoZ (figure 5) facilitates development of tool specialisations and instances integrated with different formal analysis tools. The *AUtoZ framework* component is the framework common to any UML+Z tool instantiated from the AUtoZ framework, and comprises two Eclipse plug-ins: *common* provides the necessary Eclipse features (dialogues, wizards, tools and launchers),

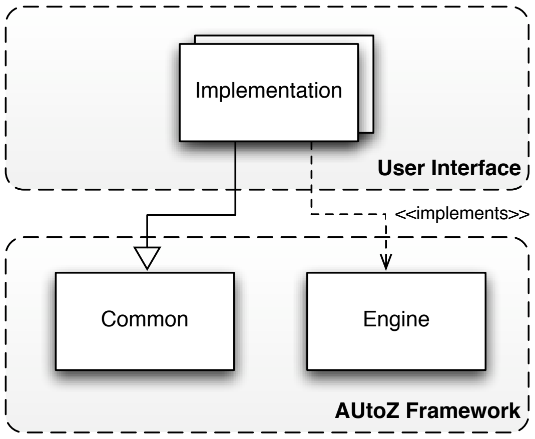


Fig. 5. The generic architecture layers of AUtoZ

and *engine* defines the interfaces that specialisations must deploy. The *user interface* component represents the specific tool implementation, also developed as Eclipse plug-ins. This allows different versions of AUtoZ using different formal tools to co-exist in the same Eclipse installation.

The activities required to transform UML to Z LaTeX markup, and to present the LaTeX to the formal tool are designed to be completely transparent. This is necessary to support the *basic user*, a developer who is well-versed in diagrammatic modelling but has little interest in formal methods. For such a user, the AUtoZ tools just extend conventional modelling tools with press-button validation of models, with no exposure to the formal underpinning.

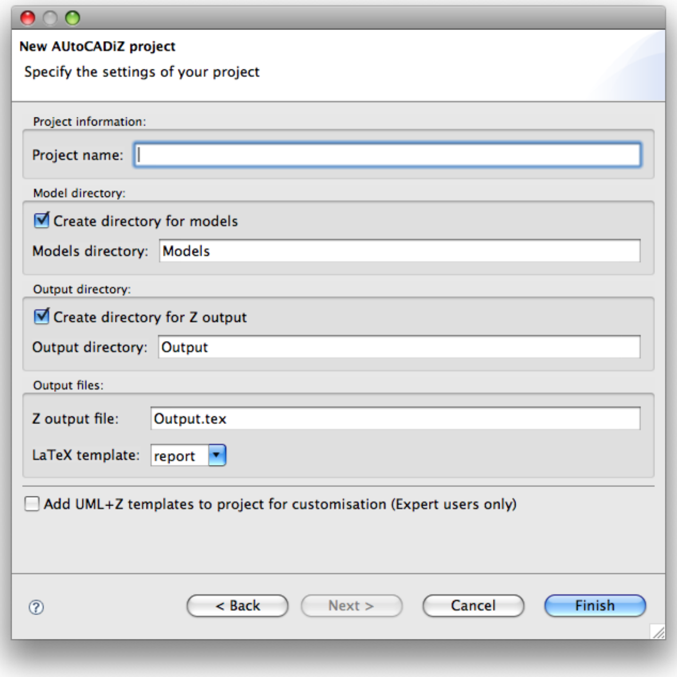
For the *expert user*, the AUtoZ tool creates an AUtoZ Eclipse project to manage files and support creation and editing of new FTL (EGL) templates. The expert user needs to be familiar with formal methods and able to interact directly with formal notations and tools, in order to develop new FTL templates and meta-theorems and to write and edit EGL transformation rules corresponding to the FTL templates – perhaps to capture the variant semantics of particular source models. In some circumstances, an expert user may be needed to complete validation, for instance where a particular model has a property that is not captured in a meta-theorem or cannot be discharged automatically.

4 Using AUtoCADiZ

AUtoZ is a generic tool framework. To illustrate its use, we consider an instance of AUtoZ that uses the CADiZ Z tool. We describe both the basic and expert uses. Space does not allow a detailed analysis, so we simply present a usage scenario for both the basic and the expert user; further examples are considered in [31].

4.1 Basic Use to Validate a UML2 Model

AUtoZ implementations are installed in the Eclipse IDE, and can be selected from the AUtoZ folder. A basic user wishing to validate the bank model shown in figures 2 and 3 would use the AUtoCADiZ implementation of the tool as described in the following workflow.


New AUtoCADiZ project

Specify the settings of your project

Project information:

Project name:

Model directory:

☑ Create directory for models

Models directory: Models

Output directory:

☑ Create directory for Z output

Output directory: Output

Output files:

Z output file: Output.tex

LaTeX template: report

☐ Add UML+Z templates to project for customisation (Expert users only)

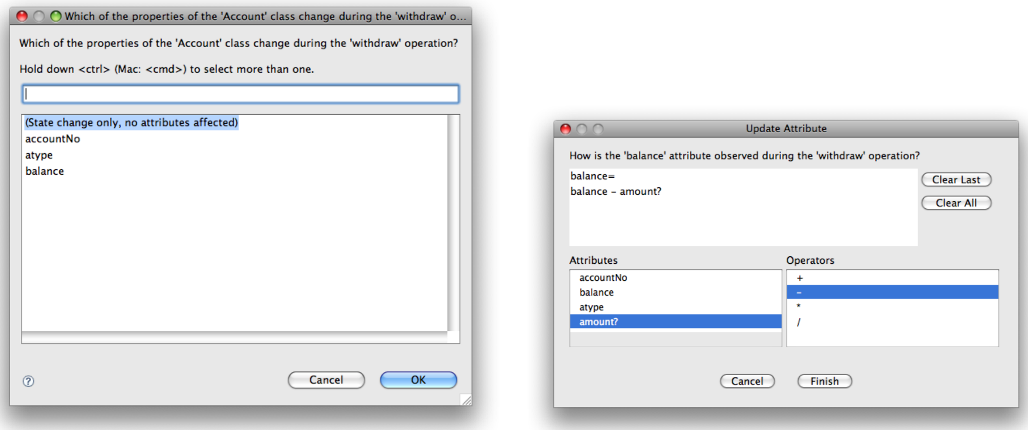
Fig. 6. Part of the dialogue used to create a new AUtoCADiZ project.

Start a new project The user opens the Eclipse IDE, and uses the `File` menu to open a `new project` dialogue. An AUtoCADiZ project is set up and named, in normal Eclipse style. For example, the user might name the project `Spec1`, and save the generated Z LaTeX markup to file `Spec1.tex`. Completion of the set-up (clicking `Finish`) results in the new project appearing in the Eclipse `Project Explorer` view. A screenshot from this dialogue is shown in figure 6.

Diagram creation AUtoCADiZ is linked to the Eclipse UML2 plug-in, which the user can use to create diagrams such as figures 2 and 3. Some design conventions have to be followed to facilitate the transformation, notably the definition of attribute types (used to generate Z given sets and other user-defined types). At this stage, the state diagram is associated to the class diagram *Account* class by naming conventions only, and the transition labelling is simply a list of the operations responsible for each change of state. On completion, the user saves the diagrams to the `Models` directory.

Create a run configuration Using the Eclipse `Run Configurations` dialogue, the user selects the `AUtoCADiZ Template` option and `New launch configuration`, associating these to the `Spec1` project and `Spec1.tex` output file. The user opts to print the CADiZ output to the console. This run configuration is saved.

Execute the formalisation Transformation is initiated by running the saved run configuration. Almost all of the target Z model is generated directly from the serialised output of the diagramming tool, with no user interaction. However, as in GeFoRME, the Z model of class operations requires detail not in the prepared diagrams. Currently, the tool presents a simple dialogue (figure 7) to guide


Which of the properties of the 'Account' class change during the 'withdraw' o...

Which of the properties of the 'Account' class change during the 'withdraw' operation?

Hold down <ctrl> (Mac: <cmd>) to select more than one.

(State change only, no attributes affected)
accountNo
atype
balance

Update Attribute

How is the 'balance' attribute observed during the 'withdraw' operation?

balance=
balance – amount?

(a) Specifying which attributes change (b) Specifying how an attribute changes

Fig. 7. Part of the user dialogue for input of details for operations. Attributes are taken from the class diagram. The change formula options reflect the Z types of the selected attributes.

the user in adding the required operation details (see section 5.1 for proposed extensions).

The tool then completes the generation of the Z LaTeX markup, which is passed to CADiZ for type-checking and analysis. The user receives the output from CADiZ at the console, as requested. Amálio has shown that his templates and meta-theorems produce type-correct Z when run on a syntactically-correct and consistent UML model [3]. In most cases, therefore, the basic user can expect to get a simple confirmation message from CADiZ. We return to this in section 5.1 below.

4.2 Expert Use: Working with Templates

An expert user needs some understanding of the template languages (EGL, FTL), and the formalisation process. The AUtoZ tools then provide an interface for adding and changing templates. The dialogue to create a project (outlined for AUtoCADiZ, above) provides the appropriate options, which result in a `Templates` directory being made available for the project, in the Eclipse `Project Explorer` view. Templates can be edited and added to subfolders containing EGL transformation rules.

For example, consider the addition of a new template which lists all of the classes in the system, named `AllClasses()`. When creating a new AUtoZ project, the expert user selects an option to include the template catalogue in the project directory. A file is added to the catalogue containing the new EGL template:

```
    [* Template to list all classes *] [% operation AllClasses
        (){
    for (c in Class.allOfType){ %]
      [%=c.name%] \\
      [%]}
    }%]
```

This template can be used in an instantiation by editing the EGL logic file in the template catalogue (`Logic.egl`) to include the following:

```
...
    [* import new operation *]
    import 'AllClasses.egl';
...
```

```
    ...
    \chapter{[%=p.name%] Package}

      [* Call the new operation *]
      \section{List of Classes in [%=p.name%] Package}
        AllClasses(); ...
```

To use the modified logic file, the expert uses the `run configuration` dialogue to select the customised file, in place of the default template catalogue (stored in the Eclipse plug-in bundle).

Further examples of expert use, including template creation, tool customisation, and tool instance creation can be found in [31].

5 Discussion

Development of the AUtoZ tool is work in progress. To date, the template instantiation of Amálio's bank case study has been completely replicated in the AUtoCADiZ tool, and parts have been replicated in AUtoZ/Eves, including the verification of instantiated conjectures and meta-theorems. In this section, we outline a number of future directions for this work.

5.1 Research Areas

Two aspects of the tool support require further research, namely the formal specification of operations from UML diagrams, and the handling of messages from the formal tools.

Operations Formally specifying UML operations in Z is not a straightforward task, because the details needed to instantiate templates are scattered across UML diagrams, require interpretation from diagrams, or are simply not amenable to recording in diagrams. The current approach using user dialogues does not scale – even modest UML diagrams often have very large numbers of operations. Further research is needed to find patterns and commonalities in operation construction that can be used to automatically incorporate operation details from other diagrams and minimise user interaction. Furthermore, the tools, and Amálio's UML+Z approach, need extending to exploit pre- and post-conditions written in OCL – an existing body of work on the use of OCL in formalisation (with B or Object-Z target models) [13,28,23] provides a starting point.

Error Handling As noted above, the AUtoCADiZ tool currently outputs CADiZ error messages direct to the user. This is unsatisfactory, as the interpretation of typical CADiZ messages requires some expertise both in CADiZ and in Z. Clearly, the tools cannot truly accessible to modellers and software engineers until the error messages generated by the Z tools can be related back to the templates, and thus to the inconsistent parts of the UML models.

In general, relating the error messages to diagrammatic models is the most important hurdle to overcome if non-expert practitioners are to be able to use formally-underpinned tools to analyse diagrammatic models. One direction would be to develop another intermediate language into which formal tool messages can be translated, and have the intermediate language manage the mapping back to diagrammatic model components. This task is complicated by the diverse ways in which formal tools present messages. There is some new work on traceability and message generation using Epsilon, which may offer a way forward [18].

5.2 Extensions to the Tool

Automatic template translation In describing the tool, we have shown that FTL maps readily on to EGL, and we note that an expert user involved in template maintenance would need to be familiar with both languages. FTL is also needed for meta-theorems and proof work, because of its formal underpinnings [3]. We have the groundwork for an automatic translation from FTL to EGL; when implemented it will allow the expert user to maintain templates using either FTL or EGL.

Template catalogue Williams [31] discussed various necessary improvements to the cataloguing of Amálio's FTL templates, but the tool currently relies on copying the templates to the AUtoZ project. A better solution would be to have a central, physically browsable catalogue, and annotated templates, as well as ways to add custom templates to the general catalogue. Note that it is not enough to provide an interactive catalogue; we need to provide support for template-validation, using Amálio's approach to meta-theorems and the relevant formal analysis tools.

Examples and help Whilst the AUtoZ tools present a solution to the shortage of formally-trained software engineers, potential basic users of the tool still need to understand how to validate models with the tools. We would like to add example models and Z specifications as training and reference literature. The Eclipse Examples Project (`www.eclipse.org/examples`) could host such examples and help for new tool users.

Improved run configuration The current run configuration does not provide suitable checkpoints in the process of generation and analysis of formal models. For example, the user has to generate a complete Z specification and pass it to the formal tool for analysis; if the `run configuration` has not been set up properly, and the connection to the formal tool is not achieved (for example, an environment variable has not been set), then the user has to start again and re-generate the Z specification. This could be addressed by providing some checking of the `run configuration` file before execution. Furthermore, generated Z models could easily be saved to file before being passed to the formal analysis tool. A logical next step would then be lazy re-generation, which only re-generates parts of models that have changed. Epsilon tools can be used to compare models, though some further work is needed to identify the FTL template connotations of changes to diagrams.

Inclusion of Amálio's templates for inheritance and composition Several parts of UML+Z are not yet included in the AUtoZ tool template repertoire. In particular, Amálio considers various semantics of inheritance [3] and proposes templates for the formalisation of composition relations [5], including the proof that models remain consistent on deletion of a composed class.

Linkage to CZT The Community Z Tools (CZT) initiative (`czt.sourceforge.net`) offers a range of support tools for Z, including parsers, translators, type-checking and a Unicode markup for Z. It would be useful to investigate integration of AUtoZ tools into this formal project.

Generalising tool support Amálio [3] developed a generic formalisation framework, GeFoRME, of which UML+Z is one specialisation. The AUtoZ tool could be made more generic by converting the components of the *AUtoZ framework* layer to a *GeFoRME Framework*, upon which any GeFoRME Framework application could be built. Under this development, AUtoZ would be constructed as a specific instance of the GeFoRME framework, with AUtoCADiZ, AUtoZ/Eves as implementations. Other GeFoRME frameworks, such as a UML+B Framework, could be built upon the same structures. Such a product-line architecture would allow massive reuse of components and powerful tools. There is also the potential to generalise the handling of source models.

6 Conclusion

The AUtoZ tools are a practical contribution to the verification, through formal analysis, of diagrammatic models. The work exploits the component-based UML+Z formalisation of Amálio, itself motivated by the desire to make formal analysis accessible to diagrammatic modellers. We have described the tools, the levels of use that they support, and the potential for generalisation and extension. Working within the Eclipse IDE and an established MDD tool suite, makes this style of formalisation both practical and usable in state-of-the-art diagram-based software modelling, and in model-driven engineering.

References

[1] Aagedal, J., A. Berre, R. Gronmo, J. Neple and T. Oldevik, *Toward standardised model to text transformations*, in: *Model Driven Architecture Foundations and Applications*, LNCS **3748** (2005), pp. 239–253.

[2] Abdul Sani, A., "Formal Analysis of Metamodel: an evaluation of an approach using ZOO templates to derive a Z model of part of EMOF," Master's thesis, Computer Science, University of York (2007).

[3] Amálio, N., "Generative frameworks for rigorous model-driven development," Ph.D. thesis, Dept. Computer Science, Univ. of York (2007).

[4] Amálio, N. and F. Polack, *Comparison of formalisation approaches of UML class constructs in Z and Object-Z*, in: *ZB 2003*, LNCS **2651** (2003), pp. 339–359.

[5] Amálio, N., F. Polack and S. Stepney, *Modular UML semantics: Interpretations in Z based on templates and generics*, in: *FACS'03 Workshop* (2003), pp. 81–100, technical Report 284: `www.iist.unu.edu/newrh/III/1/docs/techreports/report284.html`.

[6] Amálio, N., F. Polack and S. Stepney, *Formal proof from UML models*, in: *ICFEM04: Formal Methods and Software Engineering*, LNCS **3308** (2004), pp. 418–433.

[7] Amálio, N., F. Polack and S. Stepney, *An object-oriented structuring for Z based on views*, in: *ZB 2005*, LNCS **3455** (2005), pp. 262–278.

[8] Amálio, N., F. Polack and S. Stepney, *A formal template language enabling metaproof*, in: *FM 2006: Formal Methods*, LNCS **4085** (2006), pp. 252–267.

[9] Amálio, N., F. Polack and S. Stepney, *Frameworks based on templates for rigorous model-driven development*, in: *IFM 2005 Doctoral Symposium*, ENTCS **191** (2007), pp. 3–23.

[10] Amálio, N., F. Polack and J. Zhang, *Autonomous objects and bottom-up composition in ZOO applied to a case study of biological reactivity*, in: *ABZ2008*, LNCS **5238** (2008), pp. 323–336.

[11] Bast, W., A. Kleppe and J. Warmer, “MDA explained: the model driven architecture: Practice and promise,” Addison-Wesley, 2003.

[12] Bowen, J. P. and M. G. Hinchey, *Seven more myths of formal methods*, IEEE Software **12** (1995), pp. 34–41.

[13] Broda, K., D. Roe and A. Russo, *Mapping UML models incorporating OCL constraints into Object-Z*, Technical report, Imperial College London (2003).

[14] Carrington, D. and S.-K. Kim, *Using integrated metamodeling to define OO design patterns with Object-Z and UML*, in: *Asia-Pacific Software Engineering Conference* (2004), pp. 528–537.

[15] Carrington, D. and S.-K. Kim, *A tool for a formal pattern modeling language*, in: *ICFEM*, LNCS **4260** (2006), pp. 568–587.

[16] Davies, J. and J. Woodcock, “Using Z: Specification, Refinement, and Proof,” Prentice-Hall, 1996.

[17] Docker, T. W. G., R. B. France and L. T. Semmens, *Integrated structured analysis and formal specification techniques*, The Computer Journal **35** (1992), pp. 600–610.

[18] Drivalos, N., K. J. Fernandes, D. S. Kolovos and R. F. Paige, *Engineering a DSL for software traceability*, in: *Software Languages Engineering 2008*, LNCS **5452** (2009), pp. 151–167.

[19] D’Sousa, D. and A. C. Wills, “Object Components and Frameworks with UML: the Catalysis approach,” Addison-Wesley, 1998.

[20] Gamma, E., R. Helm, R. Johnson and J.Vlissides, “Design Patterns: Elements of Resusable Object-Oriented Software,” Addison-Wesley, 1995.

[21] Hall, A., *Seven myths of formal methods*, IEEE Software **7** (1990), pp. 11–19.

[22] Jackson, M., *Formal methods and traditional engineering*, The Journal of Systems and Software **40** (1998), pp. 191–194.

[23] Kleppe, A. and J. Warmer, *Object-Z to OCL dictionary*, www.klasse.nl/ocl/oz-ocl-mapping.pdf.

[24] Kolovos, D. S., “An Extensible Platform for Specification of Integrated Languages for Model Management,” Ph.D. thesis, Computer Science, University of York (2008).

[25] Kolovos, D. S., R. F. Paige and F. A. C. Polack, *Epsilon development tools for Eclipse*, in: *Eclipse Summit* (2006).

[26] Kolovos, D. S., R. F. Paige, F. A. C. Polack and L. M. Rose, *The Epsilon Generation Language*, in: *EC-MDA*, LNCS **5095** (2008), pp. 1–16.

[27] Le Charlier, B. and P. Flener, *Specifications are necessarily informal or: Some more myths of formal methods*, The Journal of Systems and Software **40** (1998), pp. 275–296.

[28] Ledang, H. and J. Souquières, *Integration of UML and B specification techniques: Systematic transformation from OCL expressions into B*, in: *APSEC ’02* (2002), p. 495.

[29] Schmidt, D. C., *Model-driven engineering*, IEEE Computer (39), pp. 25–31.

[30] Smith, G., “The Object-Z Specification Language,” Kluwer, 2000.

[31] Williams, J. R., “AUtoZ: Automatic formalisation of UML to Z,” MEng thesis, Computer Science, University of York (2009), www.jamesrobertwilliams.co.uk/publications/WilliamsMEng.pdf.