Critic Authoring Templates for Specifying Domain-Specific Visual Language Tool Critics

Norhayati Mohd.Ali[1], John Hosking[1], Jun Huh[1] and John Grundy[1, 2]
[1]Department of Computer Science and [2]Department of Electrical and Computer Engineering
University of Auckland,
Private Bag 92019, Auckland, New Zealand
nmoh044@aucklanduni.ac.nz, {john, jhuh003, john-g}@cs.auckland.ac.nz

Abstract

In recent years we have observed the extensive evolution of support tools that work with the user to achieve a range of computer-mediated tasks. One of these support tools is the critiquing system (also known as critics*). Critics have evolved in the last years as specific tool features to support users in computer-mediated tasks by providing guidelines or suggestions for improvement to designs, code and other digital artifacts. While critic tools have been demonstrated to be effective in providing feedback, critic* authoring *continues to be a big challenge. We describe a visual* design critic *authoring template approach that facilitates the construction of critics for Marama-based domain-specific visual language tools. Our template approach provides end-users and tool designers with a new way to express design critics in a natural and efficient manner. We describe prototype tool support for specifying and realizing these design critics in Marama-based tools.*

1. Introduction

The term "critic" was initially used by Miller [13] to describe a software program that critiques human-generated solutions. These types of program also known as critic tools, have evolved in recent years to support users in computer-mediated tasks by providing guidelines or suggestions for improvement [3, 4, 17, 20]. Examples of critic tools are ArgoUML [3], Design Evaluator [14] and Java Critiquer [16]. These tools were developed for the domains of UML (Unified Modeling Language), design sketching, and Java programming respectively. For instance, ArgoUML recognizes the elements and relations of UML and can advise the designer when a software architecture diagram violates the UML rules [3]. The Design Evaluator supports designers with critical effective feedback and gives reasoning on the design sketches [14]. Likewise, the Java Critiquer detects statements in a student program code that can be enhanced for readability and best practice [16]. Several studies have reported the benefits of applying such critic tools [3, 4, 13, 14, 16, 17, 20]. Among observable benefits, such tools offer proactive design feedback to users to help improve artifacts, proactively detect inconsistency and incompleteness in the design, and help users avoid careless mistakes.

Critic tools have been demonstrated effectiveness in providing these sorts of feedback. However, there has been little discussion of *critic authoring* i.e. the design and development of these critic features for tools. Critic authoring continues to be a large challenge [9, 15, 16]. Several researchers have explored different approaches to critic authoring. For instance, Qiu and Riesbeck [16] investigated how users can construct critiquing rules. Their Java Critiquer tool integrates authoring with critiquing system, to allow a teacher to check or modify the critiques generated by the Java Critiquer. Robbins and Redmiles discuss an architecture for integrating critics into design tools but require hard-coded approaches to implement their critics [18, 19].

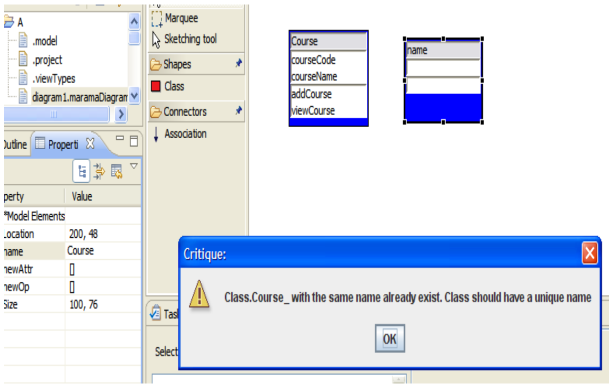
The aim of our paper is to describe a new approach using *visual critic authoring templates* to support tool and end user designers in specifying design critics for Marama-based domain-specific visual language tools. Our research is intended to extend the capability of Marama meta-tools to enable such design critics to be much more easily specified and realized by tool developers by incorporating such a visual critic authoring support feature. We first introduce the background and motivation for this research. We then present our approach to critic construction in Marama-based tools. Next, we describe our critic authoring template approach for specifying critics. An example illustrating the use of critic authoring templates is also shown. The design and implementation of visual critic authoring templates are described. We discuss the potential benefits and limitations of the critic authoring templates then summarise the research work and describe our next steps.

2. Background and Motivation

Consider a software designer developing a UML design for a software system. As they develop this design a number of "issues" may arise:

- They may construct invalid UML designs e.g. classes with the same name, classes with same-named fields, classes with invalid relationships to other classes e.g. inherit from its own sub-class;
- They may construct incomplete UML designs e.g. class with no relationship to other classes, or class method with argument missing type;
- They may construct designs which are sub-optimal e.g. fail to use appropriate design pattern or wrongly use a pattern, construct an over-complex class, repeat relationships between classes, or fail to hide certain class members.

A *design critic* feature in the UML design tool the designer is using would provide feedback to the designer on these sorts of issues. Some of these "critics" would immediately let the designer know of critical problems e.g. same-named class members or other invalid constructs. For example, in Figure 1 a critic has detected an invalid UML design construct – renaming a class to a name already in use - and is proactively informing the designer [1]. Other critics may inform the user in less obtrusive ways e.g. a list of "suggestions" in a separate viewing pane.



perty	Value
*Model Elements	
Location	200, 48
name	Course
newAttr	0
newOp	0
Size	100, 76

Class.Course_ with the same name already exist. Class should have a unique name

Figure 1. Simple critic (same named class) violation.

We have been developing the Marama [7] set of meta-tools for design and implementation of domain-specific visual language tools and wish to easily add such design critic definitions to Marama tool specifications. To date we have implemented such critics using low-level Java event handler code and OCL constraints. Such approaches are very difficult for tool developers, particularly novices, to understand and use. They are also difficult to maintain, extend and reuse. Ideally we want a visual specification tool for authoring and generating Marama design critic implementations. This would then fit well with the other visual meta-tools we have developed for the Marama platform.

Critic tools have been developed in domains such as software engineering design tools, medical information systems, computer-based education systems, and programming support tools. These existing critic tools use a variety of approaches such as rule-based, knowledge-based, pattern matching and object constraint language (OCL) expressions in their design and realization of critics. Rule-based approach consists of a condition and an action. If the condition is true, then the action is performed [15]. Actions can include suggestions, explanations, argumentations, messages or precedents of problems. For instance, ABCDE-Critic [20] uses rule-based expression to specify critics that comment on UML class diagram-based designs. The critic tool invokes critics when a condition clause is found to be true in the current design parts alerting user that the design may have problem [20]. Rules may be coded in Java, JEOPS (*Java Embedded Object Production System*), or Prolog according to the critic type.

The IDEA (Interactive Design Assistant) tool [4] produces design pattern critics implemented with Prolog rules that are directly integrated with a knowledge base. Bergenti and Poggi [4] stated that the knowledge base of IDEA is comprised of a set of design rules, corresponding critics, and a set of consolidation rules. The rules for creating the pattern-specific critics are not easy as they requires a high-level of understanding of design patterns and detailed knowledge of the Prolog and knowledge base structures [4].

Pattern matching consists of left-hand side and right-hand side rules. For instance, the Java Critiquer that checks program code [16] used pattern matching for its automatic critiquing. A JavaML pattern is used for matching JavaML code generated from the Java parser. When a pattern is matched, its corresponding critique is added into the Java source code [16].

Another way to specify critic is to use object constraint language (OCL) expressions. We illustrate a simple example of critic authoring [1] using an OCL expression extending a UML class diagramming tool that was developed using the Marama meta-tools [7].

Critics for UML class design have been identified and formulated into the OCL expressions used by MaramaTatau [10], a Marama tool for specifying model constraints, and associated with the UML tool meta-model. These critics are then applied in the executing tool (i.e. at the model or Marama diagram level) as shown in Figure 1. The OCL expression used to implement this particular constraint in the Marama UML tool is shown below:

Class.allInstances()->forAll(c1,c2 | c1 <> c2 implies c1.name <> c2.name)

Some of the difficulties and barriers in specifying critics using such OCL expressions based on our experience with Marama include:

- OCL is not easy to understand and even harder to write [21] for many tool users and developers;
- Users who lack of knowledge of OCL will have problems in specifying critics using OCL expressions;
- Difficulty in expressing [21] meaningful critics through OCL expressions as it is hard to scale the OCL expressions for complex critics

Apart from the approaches stated above, critics can be realised through the use of programming code. For instance, critics in ArgoUML [3] are coded as Java classes. ArgoUML provides a class framework, source code templates and examples to facilitate the critic implementation process. Similarly we have implemented a number of critics in Marama-based tools using its Java code event handler mechanism.

The approaches summarised above require deep understanding of the tool platform in order to design and specify critics. In fact, the customization of critics would not be easy because it requires overall comprehension of the approach employed as well as the critic domain.

Critic rules are one of the essential components in building critic tools. According to Oh et al. [15], critic rules are written by system designers in advance and once written, the customization of the rules is not easy. However, critiquing capacity and issues may need to be altered now and then in various situations [9, 15, 16]. Oh et al. [15] reported that "*Rule authoring improves the accuracy, relevance and capacity of critiquing. It enables users to store their own rules. It is an important feature that enables systems to deal with diverse situations. Rule authoring empowers designers to participate in the system's feedback process*".

Little attention has been given to provide an authoring facility for the user to add or modify critics. Qiu and Riesbeck [16] have investigated critique rule authoring. They explored the question of how users can create critiquing rules. Their Java Critiquer tool integrates authoring with a critiquing system, so that a teacher can check or modify the critics in addition to the feedback that Java Critiquer generates [16]. Some of the tools that allow for customization of critic rules are ArgoUML, IDEA, Design Evaluator, and ABCDE-Critic. For instance, ArgoUML [3] provides a class framework, source code templates and examples to support critic implementers. Authoring a new critic requires selecting a starting template, filling in relevance and timeliness attributes, coding analysis predicates and writing a headline and brief description [3]. In IDEA [4], the engineer can provide new patterns and new rules to select and fire new critics. Similarly, the Design Evaluator [14] that allows the end user (designer) to inspect and edit the rule expressions which are stored in a list. ABCDE-Critic [20] also allows the user themselves to add critics to the critiquing system, through its first-order production system.

Due to the problems and barriers noted above, we see an opportunity for a visual design notation to represent critics. The need to specify and design critics in a simple way by using an easy to use, high-level language is the motivation of this research. This would also provide a new meta-tool facility for our Marama-based tools that provides a similar visual approach to its other meta-tools. We employ a domain specific visual language (DSVL) approach in our research, which has become important in many domains of software engineering and end user development [22]. DSVLs are graphical notations specially devised for specific needs and knowledge. The languages allow anyone who is a domain expert to use the visual language as an application development tool for the domain [22]. Domain specific visual languages (DSVL) are a common approach to reduce barriers to usage, and we see an opportunity to develop a visual critic authoring framework to support software tool and end user.

3. Our Approach

To ameliorate the problems identified above with current approaches to critic authoring, we have developed a prototype of visual critic authoring tool to allow tool and end user designer to construct and specify critics for Marama-based tools. Figure 2 illustrates the process of constructing and using such critics.

A tool or end user designer uses the Marama meta-tools [7] to develop a Marama-based tool (**1**). A set of core Eclipse [5] plug-ins provides diagram and model management support for Marama modeling tools. Once a Marama-based tool is defined, a tool or end user designer can specify critics for that particular tool. Critics are specified using the Marama metamodel definer views (**2**). A tool user opens or creates a new modeling project and diagrams using this plug-ins. When a diagram is created, critics for that particular tool will be applied. If a user creates a diagram that violates the design rules of that tool, then a critique will be generated to warn user about the errors in the diagram (**3**).

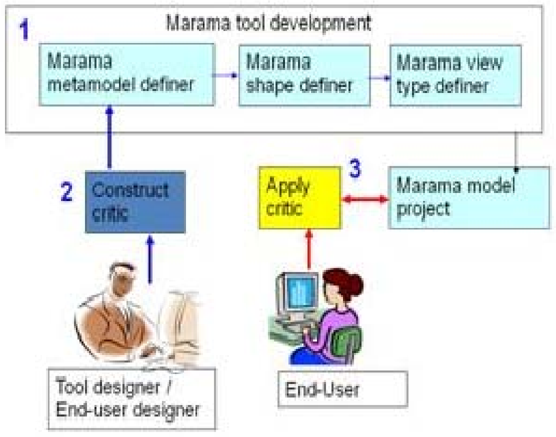


Figure 2: Marama visual critic development approach

In order to develop a critic support-based extension for Marama-based tools we have added a new functional item, *CriticShape,* to the Marama meta-model editor (refer to Figure 3). This provides tool designers with a way to add a number of critics to a tool specification and have an appropriate underlying infrastructure for the critic generated by Marama. We created a critic authoring template to allow tool and end user designers to construct appropriate critics for the Marama-based tools using a form-based approach. Critic shapes are connected to relevant tool specification elements to show users the items they are dependent on. Critic information is added to meta-model specifications (data structure entities and associations, such as classes, objects, methods and features in a UML design tool); shape definitions (visual representations of visual language elements like class, object and note shape specifications in a UML tool); and view type specifications (such as class diagram and sequence diagram definitions for a UML tool).

In the next section we describe the design of critic authoring template in specifying critics for Marama-based tools.

4. Critic Authoring Template

The idea of critic authoring templates was inspired by the business rule template proposed in the BROOD (Business Rules-driven Object Oriented Design) approach proposed by Loucopoulos and Wan Kadir [11, 23]. The details of this template can be found in [11]. The business rule templates derive from the rules definition expressed in a context-free grammar EBNF. EBNF is a meta syntax notation used to express context free-grammar: that is, a formal way to describe computer programming languages and other formal languages [6].

The business rule templates arise from a business rule typology that consists of three main types: constraint, action assertion, and derivation [11, 23]. Definitions and brief descriptions of these three types are shown in Table 1.

Table 1: Definition of constraint, action assertion and derivation (adopted from BROOD approach [11])

Type	Definition and description
Constraints	"… specify the static characteristics of business entities, their attributes, and their relationships. They can be further divided into attribute and relationship constraints. The former specifies the uniqueness, optionality (null), and value check of an entity attribute. The latter asserts the relationship types, as well as the cardinality and roles of each entity participating in a particular relationship*"*. [11]
Action assertion	"…concerns a behavioral aspect of the business. Action assertion specifies the *action* that should be activated on the occurrence of a certain *event* and possibly on the satisfaction of certain *conditions"*. [11]
Derivation	"…derives a new fact based on existing facts. It can be of one of two types i.e. computation, which uses a mathematical calculation or algorithm, to derive a new arithmetic value, or inference, which uses logical deduction or induction to derive a new fact". [11]

The rule templates are formal sentence patterns that allow the expression of business rules [11]. These templates are currently used in the business process domain for modelling. However, due to the following reasons, we have attempted to utilize these templates in the software tool domain, specifically for our prototype visual critic authoring tool [11, 23]:

- The templates use a language definition based on the context-free grammar EBNF that defines sentence patterns for rule statements;
- The templates use natural language that is easily understood to represent the rules;
- The templates provide guidance for users to determine the rules;
- The templates provide a way to construct the rule statements;
- The templates facilitate the linking of rule statements to software design elements.
- Although developed for the business domain, the templates are more general in nature and are easily adapted for use in the critic domain

Thus, we decided to adopt the templates to specify and express the critic rules. In spite of that, the critic authoring templates for the visual critic authoring tool initially covers attribute and relationship constraints only. The action and derivation part will be our next task to address. The critic rules templates that correspond to the attribute and relationship constraints are shown in Table 2.

Table 2: Attribute and relationship constraint (adopted from business rule template [11])

Attribute Constraint	<entity> must have | may have a [unique] <attributeTerm>. <attributeTerm1> must be | may be <relationalOperator> <value> | <attributeTerm2>.
Relationship Constraint	[<cardinality>] <entity1> is a/an <role> of [<cardinality>] <entity2>. [<cardinality>] <entity1> is associated with [<cardinality>] <entity2>. <entity1> must have | may have [<cardinality>] <entity2>. <entity1> is a/an <entity2>.

These critic authoring templates will support the tool and end-user designer to specify critics for Marama-based tools. Our initial attempt to utilize these templates was by specifying critics via attribute and relationship constraints. The Marama meta-tool uses *Marama Metamodel Definer* views to specify a tool meta-model. New *CriticShapes* are created and added to a *Marama Metamodel Definer* view as shown in Figure 3 (highlighted by dotted rectangles).

In Marama, a domain-specific visual language tool meta-model is expressed using an Extended Entity Relationship (EER) diagram [12] which specifies entities and relationships, together with their attributes.

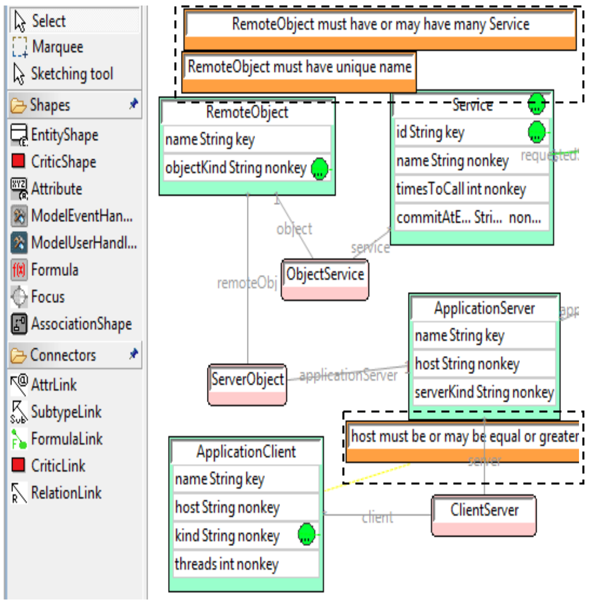


Figure 3: *CriticShape* functions added to the *marama metamodel definer view (highlighted by dotted rectangles)*

When the meta-model is equipped with sufficient information, critics can be defined via critic authoring templates. The association of critic phrase types with the corresponding tool meta-model element is shown in Table 3.

Table 3: Association of critic phrase type with the tool meta-model

Critic phrase type	Tool meta-model elements
<entity>	Entity
<attributeTerm>	Attribute
<cardinality>	end1Multiplicity, end2Multiplicity
<role >	associationEndName
<relational operator>	Enumeration
<value >	Literal value

A critic construction view interface is designed to allow tool and end-user designers to specify critics for their Marama-based tools. The interface is composed of attribute and relationship constraint properties for defining a critic. The details of critic authoring capabilities are described in the following section.

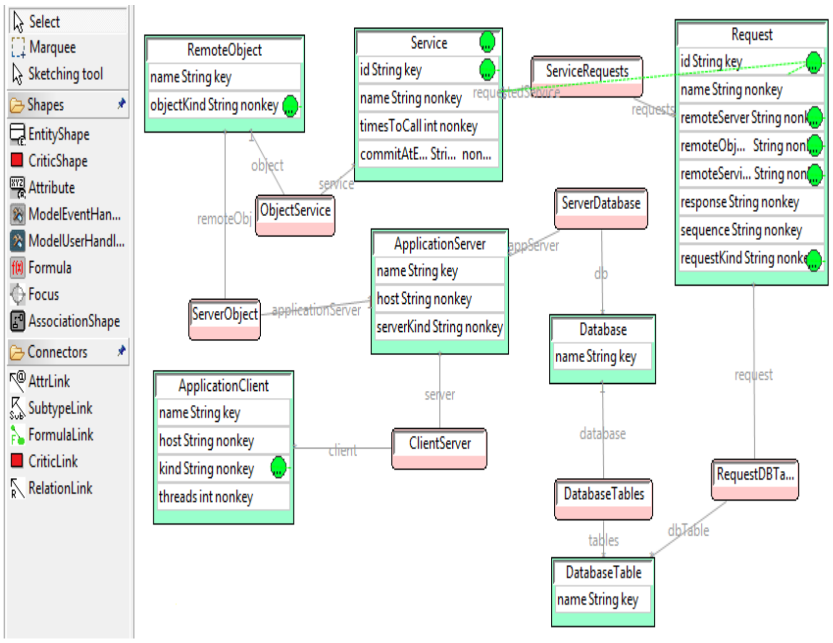


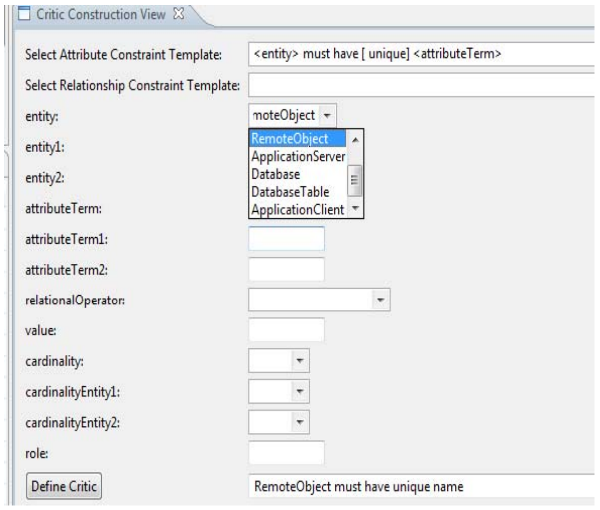
Figure 4: MaramaMTE metamodel definer view

5. Example Usage

We illustrate the use of critic authoring capabilities from one of our Marama-based tool, the MaramaMTE software architecture design tool [8]. The meta-model of MaramaMTE is shown in Figure 4. Initially a tool designer or end user designer specifies a design tool using a set of visual Marama meta-tools [7]. In this example, a tool developer has specified a variety of entities and associations to represent the structure of a software architecture e.g. clients, servers, databases, remote objects, services, requests and various relationships. The green circle annotations indicate various model constraints specified using MaramaTatau [10].

Once the meta-model, shape designs and view designs of MaramaMTE are created, the tool/end user designer is able to construct appropriate and meaningful critics for the MaramaMTE tool. Tool and end user designers who understand the domain knowledge of MaramaMTE will be able to specify critics for this tool. The *CriticShape* tool from the Marama meta-model editor is selected to define a critic. A critic construction view is then displayed to guide the critic authoring task, as shown in Figure 5. This form allows the tool developer to specify a range of critic properties based on the Business Rule concept above. These include:

- Entities and relationships in the meta-model the critic is interested in;
- Attribute(s) of entities or relationships the critic is interested in;
- Patterns to match in the design e.g. entity existence; association existence or cardinality; attribute value(s) (regular expression)
- What critic template to apply in this situation. This is a pre-defined list including uniqueness constraints, existence constraints, pattern match/non-match.



Select Attribute Constraint Template:	<entity> must have [unique] <attributeTerm>
Select Relationship Constraint Template:	
entity:	moteObject
entity1:	
entity2:	
attributeTerm:	
attributeTerm1:	
attributeTerm2:	
relationalOperator:	
value:	
cardinality:	
cardinalityEntity1:	
cardinalityEntity2:	
role:	
Define Critic	RemoteObject must have unique name

Figure 5: Critic authoring templates

In Figure 5, a critic for the RemoteObject entity is being specified using an attribute uniqueness pattern to ensure RemoteObjects have a unique name.

There are two fundamental types of constraints for specifying a critic; attribute constraint and relationship constraint. The attribute constraint consists of two templates:

1) *<entity> must have | may have a [unique] <attributeTerm>* and
2) *<attributeTerm1> must be | may be <relationalOperator> <value> | <attributeTerm2>*.

The relationship constraint comprises the following templates:

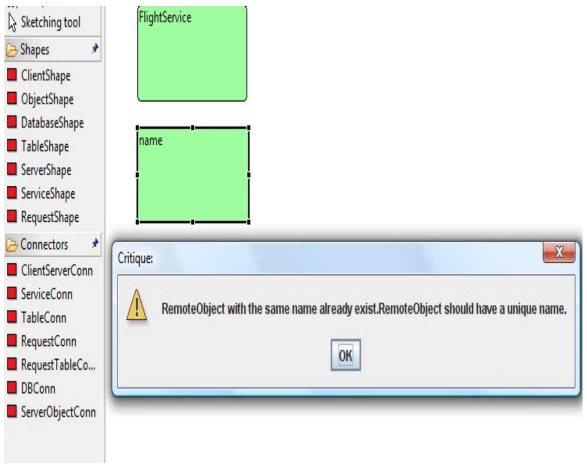
1) *[<cardinality>] <entity1> is a/an <role> of [<cardinality>] <entity2>*.
2) *[<cardinality>] <entity1> is associated with [<cardinality>] <entity2>*.
3) *<entity1> must have | may have [<cardinality>] <entity2>*.
4) *<entity1> is a/an <entity2>*.

Critics for our MaramaMTE tool, such as the one in Figure 5, are specified using these templates. For MaramaMTE critics might include completeness of the architecture design e.g. all elements linked by appropriate relationships; correctness of the architecture design e.g. no same-named services for the same remote object or same-named tables for the database; and "quality" of the architecture design i.e. checking for particular architecture styles e.g. if all services are in a single remote object; if redundancy is encountered; and so on.

After definition in our Marama meta-tool the defined critics are stored in a custom XML format in a repository along with the other tool specification elements, such as the metamodel entities and shape and view specifications.

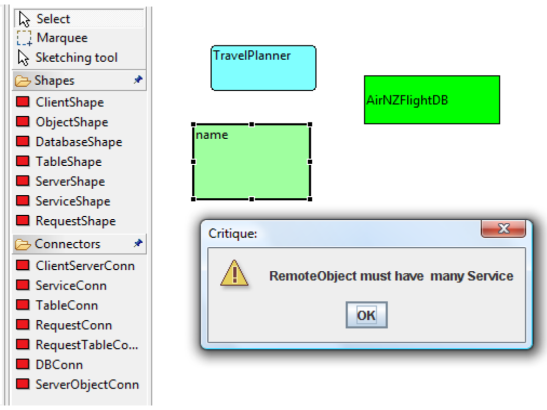
The stored critics are then applied when the tool is instantiated and executed (i.e at the model or Marama diagram level) as shown in Figure 6 and Figure 7. When Marama loads the definition of a tool it also loads the critic definitions. It then instantiates "event listeners" on the tool meta-model elements so that when these are changed, the critic "engine" is informed of the state change. The critic engine then determines which critic(s) are interested in the change and whether the critic action criteria have been met by the current state of the design. If so, the critic action is invoked e.g. message to user, message in list, highlight erroneous item(s) in a view, undo change made etc. In Figure 6 a critic detects the lack of a unique name attribute specified for a remote object, a violation of a correctness constraint. This is an example of an attribute constraint template critic. In Figure 7 a critic detects a remote object lacks service definitions. This is incompleteness in the design for the remote service. This is an example of a relationship constraint template critic.

Whenever a tool user creates or modifies one or more diagram elements that results in a violation of any design rules that were stored as critics, a critique message will be displayed to warn the user about the potential problem. These messages can also be configured to be shown in the Eclipse Problem view pane as less intrusive notifications to the designer.



RemoteObject with the same name already exist.RemoteObject should have a unique name.

Figure 6: Critic statement: remote object must have a unique name. Attribute constraint template: *<entity> must have | may have [unique] <attributeTerm>*

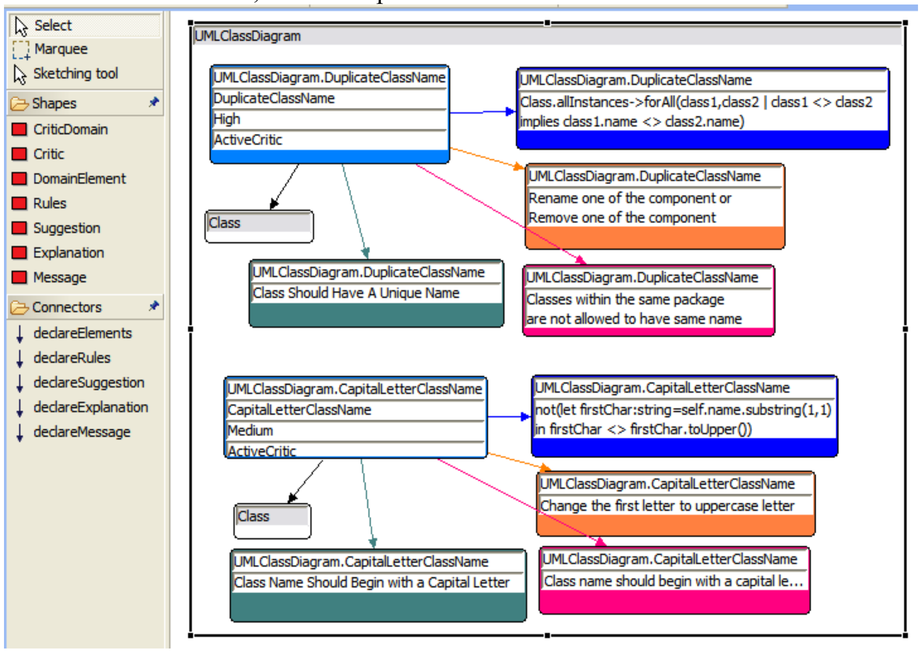


RemoteObject must have many Service

Figure 7: Critic statement: remote object must have many service. Relationship constraint template: *<entity1> must have | may have [<cardinality>] <entity2>*

We have also developed a visual representation of design critic structures (using Marama) to allow browsing of critic specifications using its own domain-specific visual language. An example of this browser in use for critics specified for the Marama UML design tool [2] is shown in Figure 8. Each critic is represented as a visual item (top left) with elements representing (counter clockwise): the meta-model entity involved;

the instantiated template; a textual description; a suggested repair if the critic is violated; and a compiled OCL form of the constraint specified.



UMLClassDiagram

UMLClassDiagram.DuplicateClassName
DuplicateClassName
High
ActiveCritic

UMLClassDiagram.DuplicateClassName
Class.allInstances->forAll(class1,class2 | class1 <> class2 implies class1.name <> class2.name)

UMLClassDiagram.DuplicateClassName
Rename one of the component or Remove one of the component

Class

UMLClassDiagram.DuplicateClassName
Class Should Have A Unique Name

UMLClassDiagram.DuplicateClassName
Classes within the same package are not allowed to have same name

UMLClassDiagram.CapitalLetterClassName
CapitalLetterClassName
Medium
ActiveCritic

UMLClassDiagram.CapitalLetterClassName
not(let firstChar:string=self.name.substring(1,1) in firstChar <> firstChar.toUpper())

UMLClassDiagram.CapitalLetterClassName
Change the first letter to uppercase letter

Class

UMLClassDiagram.CapitalLetterClassName
Class Name Should Begin with a Capital Letter

UMLClassDiagram.CapitalLetterClassName
Class name should begin with a capital le...

Figure 8: Examples of critics for a simple UML class diagram tool definition

6. Design and Implementation

We have implemented a critic support feature added to the existing Marama meta-tools. The idea of this critic support designer is to assist the tool and end user designer to construct critics for Marama-based tools. This work has added new design critic features to the Marama meta-tools and implementing a "critic engine" to load and run these critics during Marama tool use.

As shown in Figure 9, a new function, *CriticShape* was added to the Marama meta-model editor to specify a critic. Together with the *CriticShape* is the critic authoring template interface. We have developed critic authoring templates by adapting the business rule templates and designing a form-based interface to allow critic construction by tool/end user designers (refer to Figure 5). In the existing Marama meta-tools, the meta-model folder only contains association types and entity types, as shown in Figure 10. However, with the critics' definition, we have added a critic type folder as a repository to store the specified critics. Once the critics are specified and defined, these are added in the *crictictypes* folder as shown in Figure 11. Each critic is stored as an XML data file. Currently, each critic is defined based on the attribute and relationship constraint templates.

A "critic engine" loads the XML save files and instantiates an "event listener" in Marama for each of the critics defined for a Marama tool. A "critic processor" is assigned to each critic event. The event listeners receive events whenever the model is changed and determine if a particular critic is interested in the event and what action to take.

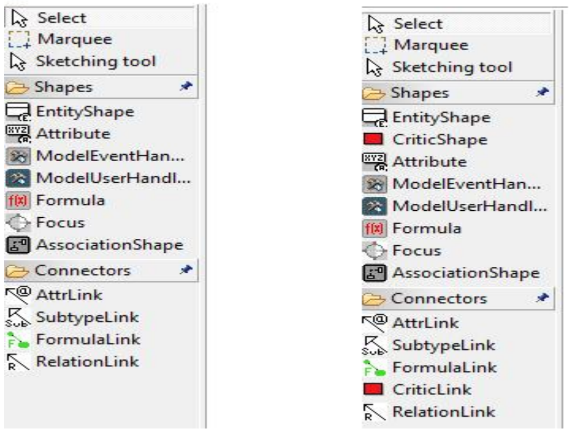
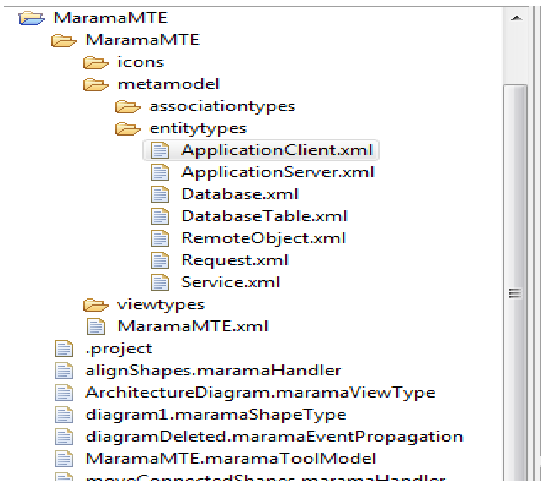


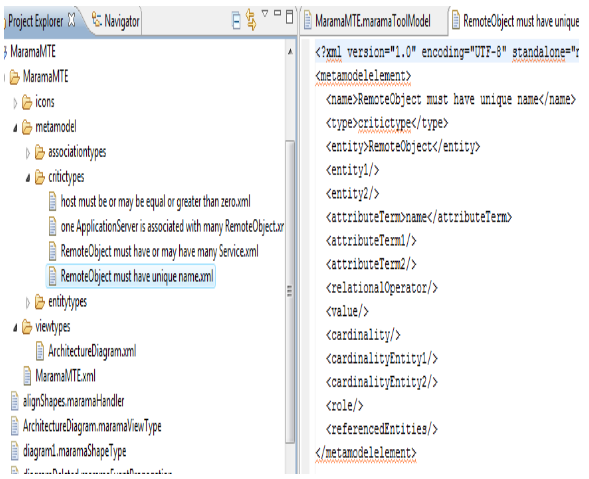
Figure 9: Old meta-model editor (left-side) and new meta-model editor (right-side) with *CriticShape* icon.

Each critic template represents a type of critic and we have implemented each critic as a concrete class. A critic processor class is instantiated using the stored XML information to determine which model element events it is interested in; patterns to match in terms of model state; and its action when receiving change events and matching part of the model state. Our architecture allows new critic templates to be added which provide users with new critics to specify for their tools.



```
MaramaMTE
  MaramaMTE
    icons
    metamodel
      associationtypes
      entitytypes
        ApplicationClient.xml
        ApplicationServer.xml
        Database.xml
        DatabaseTable.xml
        RemoteObject.xml
        Request.xml
        Service.xml
    viewtypes
    MaramaMTE.xml
  .project
  alignShapes.maramaHandler
  ArchitectureDiagram.maramaViewType
  diagram1.maramaShapeType
  diagramDeleted.maramaEventPropagation
  MaramaMTE.maramaToolModel
```

Figure 10: Meta-model folder with entity and association types



Project Explorer | Navigator

```
MaramaMTE
  MaramaMTE
    icons
    metamodel
      associationtypes
      critictypes
        host must be or may be equal or greater than zero.xml
        one ApplicationServer is associated with many RemoteObject.xr
        RemoteObject must have or may have many Service.xml
        RemoteObject must have unique name.xml
      entitytypes
    viewtypes
      ArchitectureDiagram.xml
    MaramaMTE.xml
  alignShapes.maramaHandler
  ArchitectureDiagram.maramaViewType
  diagram1.maramaShapeType
```

MaramaMTE.maramaToolModel | RemoteObject must have unique

```
<?xml version="1.0" encoding="UTF-8" standalone="r
<metamodelelement>
  <name>RemoteObject must have unique name</name>
  <type>critictype</type>
  <entity>RemoteObject</entity>
  <entity1/>
  <entity2/>
  <attributeTerm>name</attributeTerm>
  <attributeTerm1/>
  <attributeTerm2/>
  <relationalOperator/>
  <value/>
  <cardinality/>
  <cardinalityEntity1/>
  <cardinalityEntity2/>
  <role/>
  <referencedEntities/>
</metamodelelement>
```

Figure 11: Critics are stored in the critic type folder

7. Discussion

We have applied our new critic authoring approach to several Marama design tools, including MaramaEML (Enterprise Modelling Language), a Marama UML tool, MaramaMTE software architecture design tool, and MaramaVCPL, a health care plan modeling tool. We developed for each several correctness, completeness and advisory critics using our critic templates. We assessed the performance of these critics and our template approach by comparing them to OCL and Java event handler-implemented critics previously developed by hand for these tools. Our template critic extensions to the Marama meta-tools made it far easier and quicker to both develop new critics but also to modify existing critics specified for the tools.

Key benefits of our approach include the way it provides a simple way to express critic rules/phrases and resultant actions. A novice designer may easily construct and specify critics using the critic authoring templates. The critic authoring templates offers a structured form in expressing the critic rule/phrase. Marama instantiates critic rule processors when opening a tool and uses Marama's built-in event handler mechanism to proactively check changing designs.

The main limitations of this approach are that it currently only supports fairly simple design critic construction. Critics can be defined only based upon the available templates and can only pattern match a limited part of the model as supported in the template definition. Very complex critics are not able to be specified via attribute and relationship constraint templates. This is a deliberate design choice: we are aiming to support the majority of the types of critics that end users would be interested in defining, leaving the specification of more complex critics (typically done by an experienced tool designer) to OCL constraints or Java event handlers. Only limited actions are supported at present – notifying the user of critic feedback and undoing the previous editing operation. The critic engine implemented in Marama uses a simple approach to determine interested design critics which could be made more efficient when large numbers of critics exist in a tool.

We plan to extend our approach to use visual action and derivation rules as a way to specify more complex critics. Our next task is to expand the critic authoring template by considering user-specified action and derivation rules to construct such critics. More aspects of critic feedback also need to be considered. The critic authoring templates should also enable the tool and end user designer to identify and construct appropriate feedback to tool user. Some of the issues in providing critic feedback are intervention strategies [9, 15, 17], critic modalities [15], and types of feedback [9, 15, 17].

8. Summary

We have described an approach for specifying and authoring critics for Marama-based tools. Critic authoring templates adopting attribute and relationship constraint rules from the business rule templates [11, 23] were developed and added to the Marama meta-tools. We developed a prototype of this visual critic authoring template approach to demonstrate the potential of this approach to integrate into the Marama meta-tools. We have used our prototype visual critic authoring tool to demonstrate the potential of this approach by integrating critic support into the software tool development process. We illustrated the utility of visual critic authoring tool with two exemplars using critic authoring template: one for the MaramaMTE software architecture tool; the other for UML class diagram tool. We have also applied to tool to other software design tools such as business process and health care planning application design tools.

This work is very much a proof-of- concept that critic authoring templates will support the tool and end-user designer to construct and specify critics in a simple way for Marama-based tools. Our plans for future work on this research include the construction of complex critics via action and derivation rules. Furthermore, along with the critic authoring templates is the need to create critic feedback facilities. The refinement of the existing prototype will consider the enhanced use of a domain-specific language for critic specification. Evaluation of the prototype by target end users will also be performed.

9. References

1. Ali, N.M. A Generic Visual Critic Authoring Tool, Proceeding VLHCC'07, IEEE CS Press, 2007, pp.260-261.
2. Ali, N.M. Specifying Visual Design Critic Framework, Proceeding NZCSRSC'08, April 2008, Christchurch, New Zealand, pp.184-187.
3. ArgoUML, http://argouml.tigris.org/
4. Bergenti, F. and Poggi. A. Improving UML Designs Using Automatic Design Pattern Detection, In Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering (SEKE), 2000, pp. 336-343.
5. Eclipse, http://www.eclipse.org/
6. ExtendedBNF, http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf
7. Grundy, J.C., Hosking, J.G., Huh, J. and Li, N. Marama: an Eclipse meta-toolset for generating multi-view environments, Formal demonstration paper, 2008 IEEE/ACM International Conference on Software Engineering, Liepzig, Germany, May 2008, ACM Press. See also: *https://wiki.auckland.ac.nz/display/csidst*
8. Grundy, J.C., Hosking, J.G., Li, L. and Liu, N. Performance engineering of service compositions, ICSE 2006 Workshop on Service-oriented Software Engineering, Shanghai, May 2006.
9. Irandoust, H. 2006. Critiquing systems for decision support. DRDC Valcartier TR 2003-321. http://pubs.drdc.gc.ca/PDFS/unc44/p524782.pdf.
10. Liu, N., Hosking, J.G. and Grundy, J.C. MaramaTatau: Extending a domain specific visual language meta-tool with a declarative constraint mechanism, Proceeding VLHCC'07, IEEE CS Press, 2007, pp. 95-103.
11. Loucopoulus, P., and Wan Kadir, W.M.N. "BROOD:Business Rules-driven Object Oriented Design", *Journal of Database Management*, Vol.19, Issue 1, 2008, pp. 41-73.
12. Markowitz, V. Extended Entity Relationship Diagram, http://sdm.lbl.gov/OPM/DM_TOOLS/OPM/ER/ER.html
13. Miller, P. *Expert Critiquing Systems: Practice-based Medical Consultation by Computer*. Springer Verlag, New York, 1986.
14. Oh,Y., Do, E.Y.-L, and Gross, M.D., "Intellligent Critiquing of Design Sketches", in JL Randall Davis, T Stahovich, R Miller and E Saund (eds), *Making Pen-based Interaction Intelligent and Natural,* The AAAI Press, Arlington, Virginia, 2004, pp 127-133.
15. Oh,Y., Gross, M.D and Do, E.Y.-L, Computer-Aided Critiquing Systems, Lessons Learned and New Research Directions. http://code.arc.cmu.edu/lab/upload/caadria-oh.0.pdf
16. Qiu, L., and Riesbeck, C.K., "An Incremental Model for Developing Educational Critiquing Systems: Experiences with the Java Critiquer", *Journal of Interactive Learning Research*, 2008(19), pp.119-145.
17. Robbins, J.E. 1998. Design Critiquing Systems, Technical Report UCI-98-41. http://www.ics.uci.edu/~jrobbins/papers/CritiquingSurvey.pdf.
18. Robbins, J.E., Hilbert, D.M. Redmiles, D.F. Software Architecture Critics in Argo. Intelligent User Interfaces 1998, pp. 141-144
19. Robbins, J.E., Redmiles, D.F. Software architecture critics in the Argo design environment. 1. Knowledge-Based Systems 11(1), 1998, pp. 47-60.
20. Souza, C.R.B., et al. A Group Critic System for Object-Oriented Analysis and Design, In Proceedings of the 15th IEEE Conference on Automated Software Engineering, IEEE Press, 2000, pp. 313-316.
21. Sourrouille, J.L. and Caplat, G. Constraint Checking in UML Modeling, In Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), 2002, pp. 217-224.
22. Sprinkle, J., and Karsai, G." A Domain-Specific Visual Language for Domain Model Evolution", *Journal of Visual Languages and Computing*, Vol.15, Issues 3-4, June-August 2004, pp 291-307.
23. Wan Kadir, W.M.N., and Loucopoulus, P. "Relating evolving business rules to software design", *Journal of Systems Architecture*, 50(7), Elsevier, 2004, pp.367-382.