In Proceedings of 2007 IEEE Symposium on Visual Languages and Human-Centric Computing, © IEEE.

MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism

Na (Karen) Liu[1], John Hosking[1] and John Grundy[1, 2]
Department of Computer Science[1] and Department of Electrical and Computer Engineering[2], University of Auckland, Private Bag 92019, Auckland, New Zealand
{karen | john-g | john }@cs.auckland.ac.nz

Abstract

It is increasingly common to use metatools to specify and generate domain specific visual language tools. A common problem for such metatools is specification of model level behaviours, such as constraints and dependencies. These often need to be specified using conventional code in the form of event handlers or the like. We report our experience in integrating a declarative constraint/dependency specification mechanism into a domain specific visual language metatool, focussing on the tradeoffs we have made in the notational design and environmental support used. The expressive power of the mechanism developed is illustrated by a substantial case study where we have redeveloped a complex visual tool for architectural modelling, eliminating conventional event handlers.

Keywords: visual constraint language, visual language metatool

1. Introduction

It is increasingly common to use metatools to specify and generate domain specific visual design tools. Examples of such metatools include MetaEdit+ [12], GME [1], Eclipse GMF [8], and Microsoft DSL Tools [2] together with our own Pounamu[1] [20] and Eclipse-based Marama[2] [10] toolsets. High-level visual specifications of tool meta-models and visual language notations allow end users to modify aspects of their tools such as appearance of icons and composition of views and meta-models.

However, an area that commonly proves difficult for meta-tool designers is the specification of model level behaviours, such as constraints, dependencies, element initialisations, calculations, etc. Most approaches for model behaviour specifications use conventional code in the form of event handlers or constraint expressions. For example, Pounamu uses Java-based event handlers, GMF and GME use textual OCL [18] expressions, and MetaEdit+ uses a combination of constraint wizards and external code snippets. The difficulty with all of these approaches is that the resulting behavioural specifications are not strongly integrated with the visual meta-model, resulting in a variety of, in cognitive dimensions [7] terms, hidden dependency, consistency, juxtaposability and visibility issues.

In this paper we describe MaramaTatau [3], an extension to our Marama metatool set, which provides the ability to specify behavioural extensions to Marama meta-models. Although, like GMF and GME, the behaviours have an OCL formula basis, we have attempted in the environment design to mitigate the hidden dependency, consistency and visibility issues noted above. In the following section we motivate and background our work in more detail. We then describe our new approach, using a simple example to illustrate. A more detailed case study follows, showing the reengineering of a previously developed tool, in the process eliminating complex handler code. We discuss the implications of our work then summarise the results achieved and proposing further work.

2. Background and Motivation

In our prior work, we have developed a variety of frameworks and metatools to support specification and implementation of multiple view, multiple notation domain specific visual language environments [11][20]. In each of these platforms we have struggled to find an appropriate means of specifying behaviour, despite having used a variety of approaches. One, used in our JViews framework [11] and Pounamu metatool, was *escape to code* with conventional code accessing tool data structures via an API. This mechanism, also used by MetaEdit+ and DSL Tools, while very powerful is also problematic, requiring much repetitive coding and thorough end user knowledge of the metatool API. It also has significant hidden dependency, visibility and juxtaposability problems due to the differing abstraction levels involved.

A second approach, used in our BuildByWire tool [17], adopts a concrete visual specification of interface component constraints for use in our JViews

[1] *Pounamu* is the Maori word for greenstone jade
[2] *Marama* is Maori for moon, an Eclipse generator
[3] *Tatau* is the Maori word for "add up" or calculate

framework. This works well for shape and editor constraint specification. Kaitiaki [13] uses a dataflow metaphor to specify view level behaviour-oriented constraints for our Pounamu metatool. This is more abstract than BuildByWire, but uses exemplars of user interface components to make the specifications more concrete. These two metaphors do not, however, extend well for model level constraints and dependencies due to the lack of user interface exemplars to "concretise" the model level specifications, and the awkwardness of expressing calculations, common in model level constraints, using these metaphors.

An increasingly common approach is to express model level constraints as declarative formulae. MetaEdit+ uses a combination of wizards to define such constraints and natural language rendering to visualise them. GME and GMF both use OCL expressions to specify constraints and dependencies. These latter have the advantage of using a standardised and compact notation (OCL) familiar to modellers. These approaches are more successful than "escape to code", but still involve a large notational and semantic separation between the textual constraint formula and the visually specified metamodel. GME attempts to bridge this gap by annotating visual model elements to indicate constraints apply to them, but editing and understanding a constraint still presents significant hidden dependency and consistency issues.

Formulaic constraints and dependencies are common in spreadsheets [5][15]. Spreadsheet formulae permit declarative specification of system behaviours and automatic evaluation of them. A highly concrete metaphor is used, however, with the grid structure reused for both formula programming and execution, providing good preservation of the end user's mental map of the application. This approach is thus not immediately adaptable to the domain of metamodellers as there is necessarily a separation between the metamodel specification and its end user realisation as a set of view editors in a generated application. However, approaches such as ClassSheets [5] and Forms/3's prototype approach [15] provide some indication of how aspects of this metaphor could be adapted to suit the metamodelling domain. Of particular interest are hidden dependency mitigation approaches, such as dependency link views, and the ease of formula construction afforded.

We have recently developed Marama, a new metatool platform. Marama has evolved from our standalone Pounamu tool, but is implemented as a set of Eclipse plugins, leveraging Eclipse EMF [4] and GEF [6] frameworks. As part of this redevelopment, we took the opportunity to address Pounamu's difficulties in expressing model-level constraints and dependencies. Both Pounamu and Marama adopt an extended entity relationship (EER) model as the metamodel specification mechanism. The EER model contains definitions of a set of entities, relationships, and attributes. We saw a possibility to extend this simple representation with declarative constraint/dependency specifications. We were attracted to a formulaic approach but wanted to minimise/mitigate the cognitive dimensions tradeoffs involved. This led to the following set of requirements for the constraint representation mechanism:

- Aim for target end users who are programming literate and familiar with modelling concepts
- Ability to represent model level constraints, dependency calculations, and initialisations
- A compact representation
- Use of a standardised notation familiar to the target end users for accessibility of use
- Ability to minimise/mitigate hidden dependency and visibility issues between the constraint specification and the visual meta model specification
- Ability to rapidly compose constraints
- Ability to simply visualize execution behaviour

In the next section, we introduce MaramaTatau, our approach to implementing these requirements.

3. MaramaTatau

MaramaTatau is strongly focussed on structural constraints. The primary notation for constraint representation in MaramaTatau is declarative OCL expressions, a representation chosen for the following reasons:

- OCL expressions are relatively compact (certainly in comparison to Java event handler code).
- OCL is specifically designed as a language to express model level constraints. It thus has primitives for common constraint expression needs, e.g. navigation of relationships, set and list manipulation (including aggregation), and common calculation operations of various types (arithmetic, string, boolean).
- While designed for OO metamodels, OCL is equally applicable to Marama's EER metamodels.
- OCL is a standardised language, likely to be familiar to our intended end users.
- The quality of OCL implementations is increasing.

Providing an OCL expression editor, similar to those in GME and GMF, covers the first four requirements of the previous section. What differentiates our approach, however, is the way we address the other requirements. Our approach is to combine the advantages of the textual OCL formulae

with the ease of formula construction afforded by spreadsheets, together with a lightweight, yet robust mechanism to mitigate hidden dependencies.

Figure 1 shows the Marama metamodel editor with MaramaTatau extensions. The metamodel shown is for a simple aggregate system modeller, comprising wholes and parts, represented by the Whole and Part entities (1), both generalising to a Type entity and related by a Whole_Part relationship (2). The entities have typed attributes, such as name, area, and volume. Below is the formula construction view (3). This allows OCL formulae to be selected, viewed and edited. A list of available OCL functions (4) is used for formula construction. The formula shown "`self.parts-> collect( cost * (1.0 + markup))->sum()`" specifies that the price attribute of a whole is calculated by adding the products of its parts' cost and markup values.

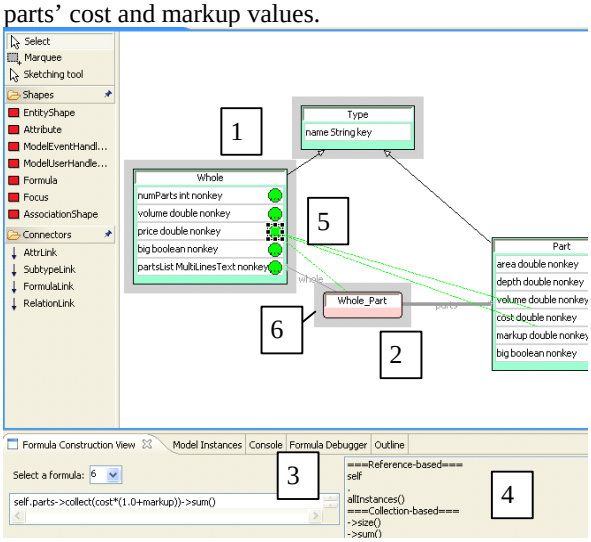

Figure 1. MaramaTatau visual notation

Also shown in the visual metamodel view are various annotations (5) indicating the presence of constraints. Coloured circles placed on attributes or entities indicate that an OCL formula has been defined to respectively calculate their value or provide an invariant constraint over them. All of the attributes of the Whole entity have such formulae, as do the volume and big attributes of Part. The annotation is coloured differently (red) if its formula is semantically incorrect. Dependency link annotations provide more detailed information about a selected formula by connecting its annotation to other elements used in the formula. For example the formula for the price of a Whole entity is selected (selection handles showing). The dependency links show that the price formula is dependent on the cost and markup attributes of the Parts connected to the Whole by the Whole_Part relationship. Entities and connection paths that are directly accessible when constructing a formula (Whole, Type, Whole_Part) have grey outline borders around them (6, see below).

We have carefully defined the interaction between the two views to enhance visibility and minimise or mitigate hidden dependency issues. Visibility and hidden dependency issues are addressed by the following mechanism:

- The OCL and metamodel editors are juxtaposed together to improve visibility
- Simple annotation of the model elements indicates formulae related to them are present and whether they are semantically correct. This is similar to the GME constraint annotations.
- Formulae can be selected via either the metamodel view annotation or from a selection list in the OCL view. This means constraints can be navigated to/accessed from either view. Selection in one view causes selection in the other.
- The dependency link annotations in the metamodel view provide, at a glance, more detailed understanding of attributes and entities used in the formula. This visualisation extends beyond that of GME, providing a more detailed, constraint specific understanding of dependencies involved. The annotations are modified dynamically as formulae are edited maintaining consistency between the views. The extra annotations are deliberately made visible only when a constraint is selected to minimise clutter, permit scalability, and provide task focussed information to the end user. This approach is similar to dependency visualisations provided in some spreadsheets, linking cells with formulae to those they depend on, but applied to a graphical modelling metaphor rather than a spreadsheet grid. Coloured dependency links and textual element references – as done in some spreadsheets – is a straightforward extension to provide even finer-grained indication of dependencies.

The rapid composition requirement is addressed by several techniques, adapted from common spreadsheet use. These assist with hidden dependency and visibility issues. Formula construction can be done either textually, via the OCL view, suitable for those highly OCL fluent, or "visually" via direct manipulation of the metamodel view and function selection list to automatically construct entity, path, and attribute references and function calls. Clicking on attributes in the metamodel view places an appropriate reference to that attribute into the formula. Path references are constructed by clicking on the relationship and then an attribute in the entity referenced by that relationship. A function selected from the list in the OCL view is inserted as a function call into the formula being edited, similar to formula selection in spreadsheets.

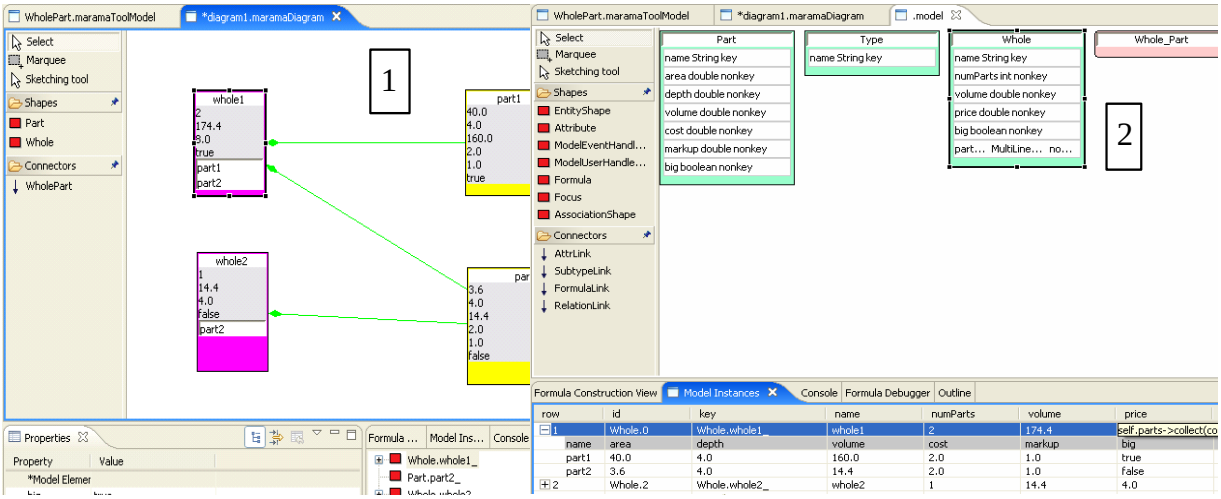

Figure 2. (1) Model instantiation view and (2) model instance view.

A difference from spreadsheet formula construction is that when constructing a formula, only certain elements are semantically sensible at a particular stage of editing whereas in spreadsheets, any cell may be referenced (circular references excepted). For example, clicking on a Part attribute, without first constructing a relationship reference via Whole_Part, does not make sense. To guide users, grey border highlighting indicates entities and relationship links valid to select at a given point in formula construction. Should a semantically incorrect formula be constructed, the annotation change in the metamodel view provides immediate visual feedback of the error.

Another area of departure from the spreadsheet metaphor is in model instantiation. In spreadsheet based systems the metaphor used is both very concrete and live. The very nature of metatools, where an abstract conceptual metamodel is defined necessarily separately from the views of that model means concreteness must be sacrificed, and hence there is an additional set of hidden dependencies and visibility issues, between the metamodel definitions (including the OCL formulae) and the model instances, created. In designing MaramaTatau's runtime implementation we have introduced several mechanisms to mitigate these hidden dependency issues. Liveness, however, is already well supported in Marama. Unlike almost all other similar metatools, Marama tool definitions can be modified on the fly, with changes immediately reflected in any open tool instances.

Figure 2 (1) shows a modelling tool based on the Whole Part metamodel used to edit an example model (the icons and connector forms, and view-model mappings are defined separately using other Marama metatools). When such a model instance is being manipulated (entities and relationships created, property values edited) relevant formulae are interpreted and the derived values assigned to their contextual model entity/relationship properties. For example the parts list in the whole1 Whole entity, represented as a multi-line list in the visual modelling view, has value `[part1,part2]` constructed using a formula that collects the name of each linked part into a new list. Properties with values defined by formulae are not editable by the end user. Clicking on a formula in the formula tree window at the bottom right causes properties calculated by that formula to be highlighted in the property window to its left.

In **Figure 2** (1), only a single Whole Part view is shown. Marama supports specification of tools with multiple views and multiple notations; each view being mapped to a common underlying model (specified using the metamodel tools). To allow end users to visualise the shared model, a model instance view is provided. **Figure 2** (2) shows an example of this view for the Whole Part model. The topmost view contains all entity and relationship types defined in the metamodel view. The same element representation is used as in the metamodel specification to minimise/mitigate hidden dependency issues between the metamodel specification and model instance view. Note that we have chosen not to replicate exactly the same view because a Marama metamodel can itself be specified across *multiple* metamodel views. The model instance view depicts the union of meta elements in *all* such views, so does not follow exactly the same layout. This is an area we are still experimenting with. An alternate approach is to provide a set of model instance views, one for each metamodel specification view.

The table view at the bottom of **Figure 2** (2) is a spreadsheet like representation of all instances of the element type selected in the top view; the Whole entity

in the view shown. Each row details attribute values for an instance of the selected entity. These rows may be expanded, as shown for the first element, to provide details of other elements associated with the chosen element via relationships. In the example shown the two Parts associated with the first of the Whole elements are detailed. This view, thus provides a rapid understanding of model elements and related values.

Formulae for calculated attributes are shown by tooltip when the mouse hovers over such an attribute value (as for *price* in **Figure 2** (2)). This mitigates the hidden dependency between the concrete value and its OCL formula. Further mitigation is provided by a formula debugger view (Figure 3). This provides a dynamic, textual visualisation of formula execution, concurrent with changes occurring in the visual views (providing good *visibility* of behavioural changes). These two features together satisfy the final requirement: to simply visualize execution behaviour.

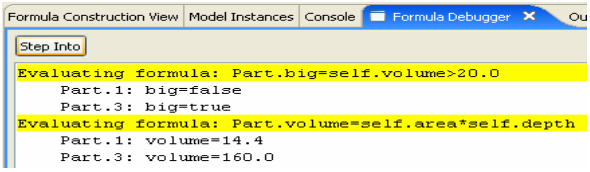

Figure 3. Formula debug view

4. Case study

The previous section introduced the notational features of MaramaTatau plus environment support mechanisms to mitigate hidden dependency and visibility issues. To evaluate the scalability and utility of the approach we present a larger case study reengineering a previously developed Marama tool to replace "escape to code" behavioural specifications with MaramaTatau constraints.

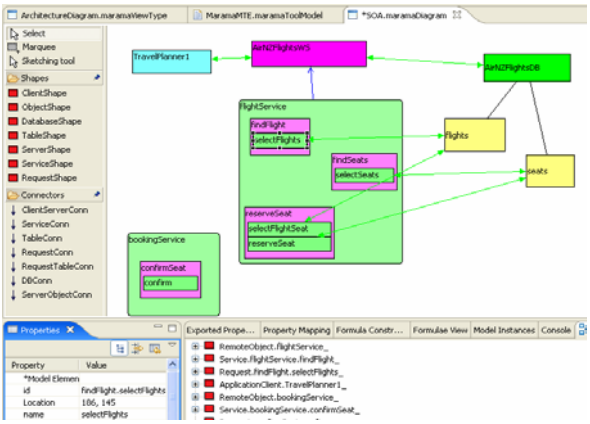

Figure 4. A MaramaMTE architecture view

MaramaMTE [9] is a complex visual tool for software architecture design and performance test-bed generation. It provides a number of notational views, including a structural architecture view and a pageflow view for specifying abstract user interface behaviour, all linked to a common underlying model. Figure 4 is a screen dump of MaramaMTE in use, with a structural architecture view describing a three-tier client-server architecture for a travel planning system shown.

In its original form, the implementation of MaramaMTE required a substantial number of java-based event handlers to implement various calculations and constraints. Consider remote objects, the rectangular icons containing other icons representing services they provide. For example the bookingService remoteObject has associated a confirmSeat service. These remote services have an id attribute which is the concatenation of the name of the remote object and the name of the service (eg bookingService.confirmSeat). The handler code implementing this simple constraint is substantial. Part of it is shown in **Figure 5**. Much of the code involved is repetitive or formulaic, manipulating Marama data structures via its API to access attribute values, calculate values, and assign results.

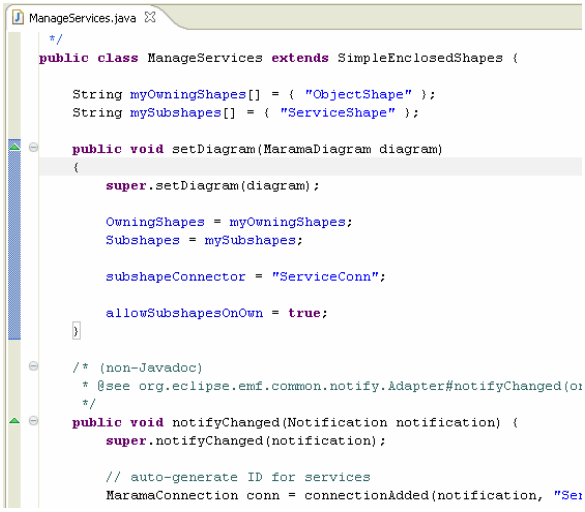

Figure 5. Handler code implementing constraint

The screen dump in the centre of Figure 6 shows a major portion of the metamodel for the reengineered MaramaMTE. A number of formulae have been defined to calculate various attribute values. Below an expanded view of the formulae list shows OCL expressions for each constraint defined. Above an expanded view of part of the metamodel shows the Service and Remote Object entities and the relationship between them plus an OCL formula for the service id (formula 8 in the list at the bottom). This expression replaces the complex handler code in **Figure 5**. This specification is not only much more compact, it is also much easier for the end user to understand and reuse.

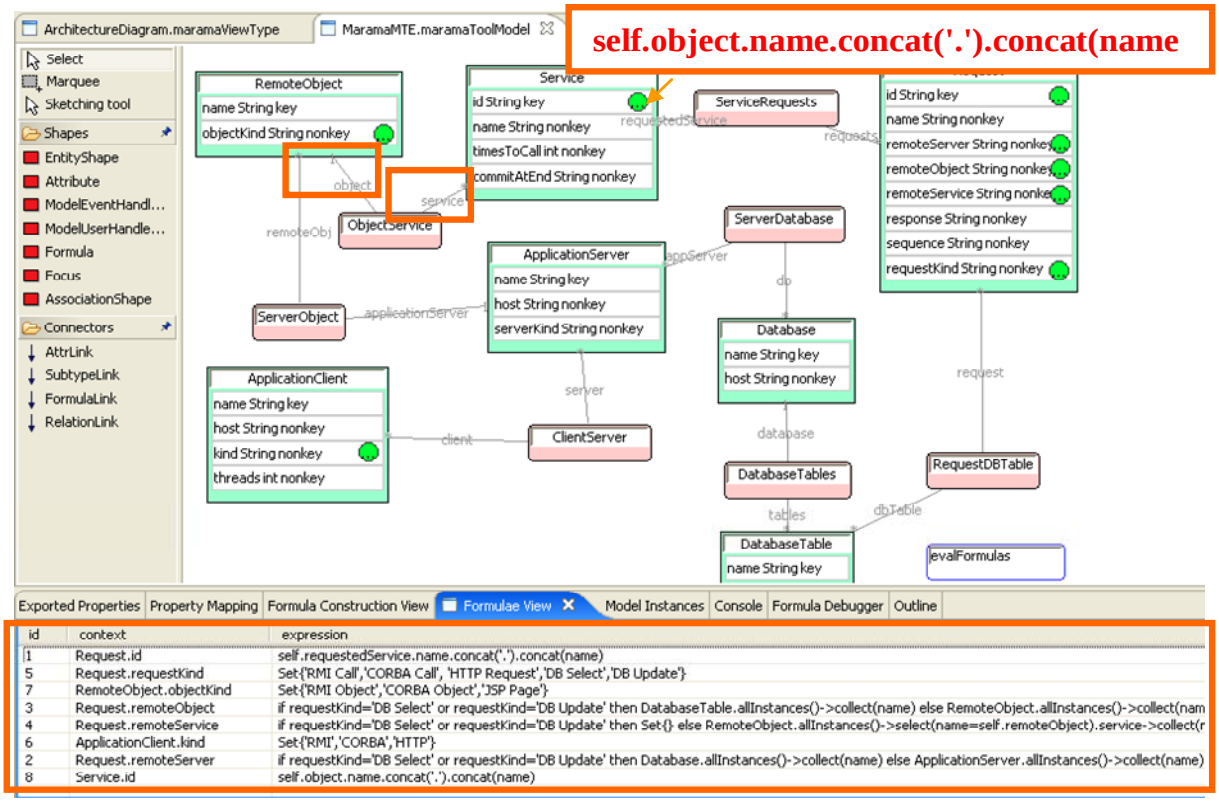

Figure 6. MaramaMTE model behaviour specification

A range of other constraint expressions are shown in the formula list at the bottom. The first of these is an id calculation for service requests similar to the remote object service id formula. The next two initialise attributes representing the types of middleware supported by the test bed generator. These are used in the modeller to constrain the combo-box values selectable by the end user. Those for the remoteObject and remoteService attributes of Request are moderately complex conditional expressions, which involve tracing a series of relationship paths to derive the names of the remote object and remote service invoking the request. These are thus derived attributes, caching values for more convenient use.

As mentioned in the previous section, formulae can also be placed on entities to specify entity invariant constraints. In Figure 7 (a) we have extended the MaramaMTE metamodel with a constraint specifying that every service instance must serve at least one service request. This is expressed as a constraint on the Service entity, with OCL expression "`self.requests->size()<>0`", shown in the overlay.

When this formula evaluates false for a service, e.g. the cancelBooking service of the bookingService remote object in Figure 7 (b), a constraint violation error is generated. In this case a problem marker is generated in the Eclipse Problems view (shown below) to provide the user details of the constraint violation. In this case, to solve the identified error, the user needs to add a request entity for the identified service. When this is done, the constraint evaluates to true and the constraint error is removed from the Problems view.

Feedback from the developers of the original and reengineered MaramaMTE applications has been very positive. Combined together the attribute calculation and invariant constraint formulae were more than adequate to eliminate all event handlers implementing model level constraints in MaramaMTE. The developers felt that the compactness and accessibility of the constraint notation and its environmental support had made the application as a whole much more easily understood and maintained. The notational mechanism also proved to be highly scalable, being unobtrusive when the tool designer's focus was on understanding metamodel structure, but providing ready ability to focus in and obtain more detailed information about particular constraints without losing the metamodel context they are situated in. The runtime support has proven more than adequate to allow tool users to comprehend the calculations being undertaken and for the tool designer to quickly debug constraints defined.

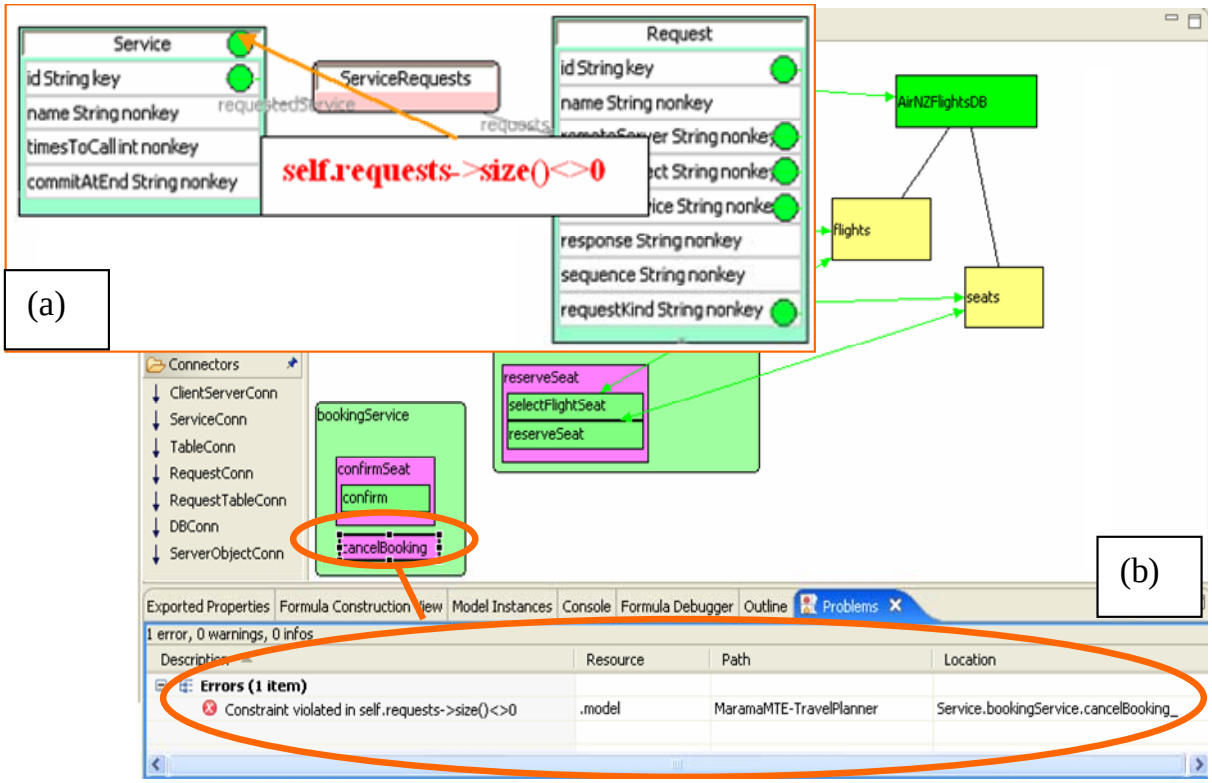

Figure 7. Using formulae to constrain entities

5. Implementation

Figure 8 shows a high-level architecture view of the Marama meta-tool and Marama Eclipse plug-ins. Marama uses Eclipse's GEF and EMF frameworks for view and model representations respectively [10]. Marama's development was bootstrapped from our earlier Pounamu metatool [20]. Originally, we developed a modeller capable of reading Pounamu tool definitions and generating Eclipse plugin implementations. The Marama metatools (to the left in Figure 8) were then initially defined using Pounamu and refined within Marama itself, including implementation of the MaramaTatau formula definition extensions to the metamodel designer.

Marama tool specifications are represented in XML format saved to tool projects (1) as hierarchically organised directories or ZIP archives. MaramaTatau formulae are stored as XML tags together with other metamodel elements. Users of Marama locate a desired existing Marama project to open or request a project be created via the standard Eclipse resource browser (2). When a project is re-opened or created in Marama, the corresponding Marama tool specification files are read and loaded into DOM objects (3). These are parsed to provide an in-memory representation of the Marama tool configuration. This tool configuration is used to configure an EMF-based in-memory model of both model and view (diagram) data (the names and properties of all entities, associations, formulae, shapes and connectors). Formulae on the user model are transformed to OCL representations on the Marama EMF model instance (4). This process is hidden from the user. To realise MaramaTatau we integrated the EMF OCL [3] framework to implement a dynamic compiler and interpreter for MaramaTatau OCL specifications. The tool configuration is also used to produce the editing controls of Marama GEF-based diagram editors (i.e. the allowable shapes and connectors, their renderings, editable attributes, etc). When a diagram is opened, Marama configures a GEF editor and renders the diagram (5). As Marama view or model data is updated, events are sent and interpreted into EMF object requests and updates, including triggering and executing relevant compiled OCL expressions (6). Marama uses EMF's XMI save and load support to store modelling project data (7) [10].

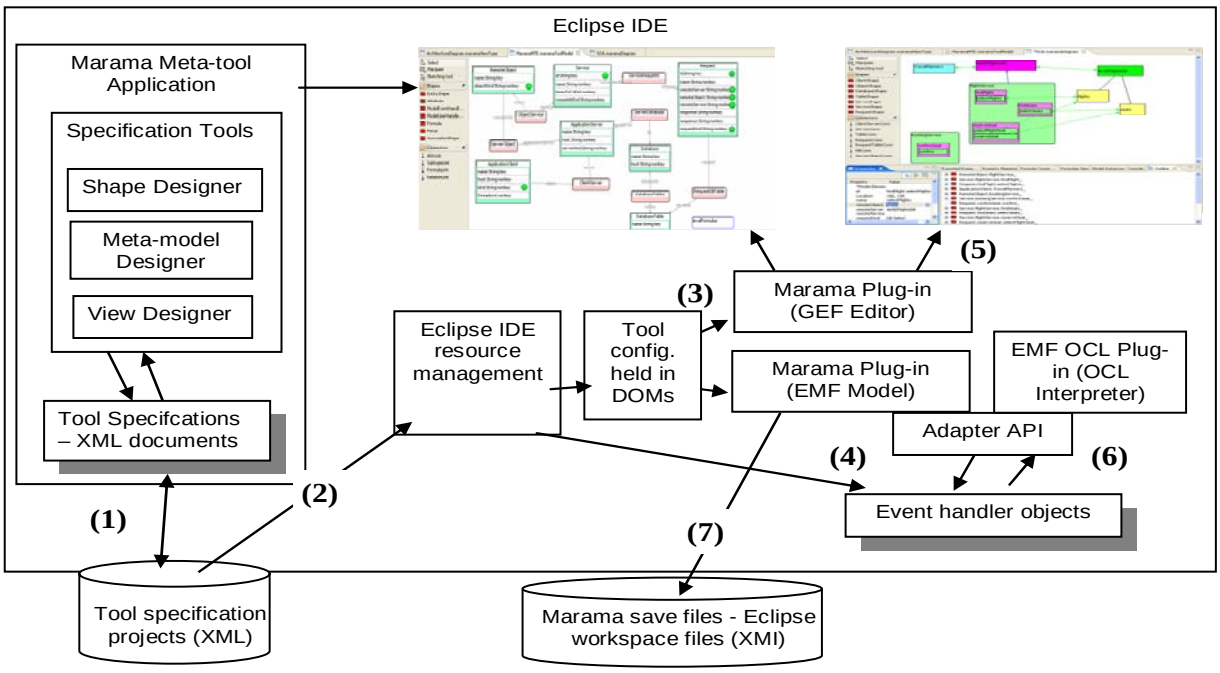

Figure 8. Implementation of MaramaTatau

6. Evaluation

The case study has demonstrated that the approach we have developed is both effective and scalable, and amply meets the requirements we established for it. Informal feedback from the case study developers has been positive. For additional feedback, we have used a focus group approach, presenting and demonstrating case studies to a small group (less than 10 participants) of experienced modellers, to gather qualitative feedback on the MaramaTatau visual notation and environment. Participants found MaramaTatau to be easy to understand and efficient to use to manage constraints and dependencies. We are in the process of performing a much more substantial evaluation (approximately 100 participants), similar to the one we undertook for our Pounamu tool [20], of the complete Marama environment, including MaramaTatau. Results of this will be presented in due course, but we have been sufficiently encouraged by our informal evaluations to include MaramaTatau in the publicly released version of our Marama tool [16].

In developing MaramaTatau, our focus has been on providing a compact and accessible constraint representation for Marama, while minimising hidden dependency, juxtaposability and visibility issues. To understand other tradeoffs that we have made to achieve our primary aims, it is useful to also evaluate MaramaTatau against other cognitive dimensions.

The visual abstractions introduced are visual iconic constructs and data dependency links between them. This is quite a terse (low *diffuseness*) extension to the existing metamodel notation and the abstractions are quite low level, providing a simple overview of constraints and dependencies, and hence have *low abstraction gradient.*

Error proneness has been reduced significantly. The existing Marama Java-based Marama event handler designer is very error-prone for both novice and experienced users due to its reliance on API knowledge and Java coding together with the numerous hidden dependencies with the visual metamodel. MaramaTatau reduces error proneness by avoiding API details and directly using concepts visible in the metamodel.

The verbosity (high *diffuseness*) of the textual OCL, due to its many built in functions, does, however, present similar opportunities for error as does API mastery. The verbosity also introduces some degree of *hard mental operations* as users must remember what function is appropriate for a given purpose. However, the relative familiarity of OCL with the target end user group mitigates this and also means good *closeness of mapping* for them. The compact nature of the representation, point and click construction, and automatic construction of the visual model annotations, means *viscosity* is low.

MaramaTatau allows *progressive evaluation* of a constraint specification via Marama's live update mechanism. Modifications to formulae take effect immediately after re-registration in an end user tool. A

visual debugger allows users to step through a formula's interpretation using the same abstraction level as they were developed in. By contrast, java event handlers require conventional java debuggers and a good knowledge of Marama's internal structure.

The MaramaTatau entity invariant formulae mechanism provides a rudimentary form of design critic [19]. In current work we are extending this approach to provide a more general critic authoring mechanism integrated with our Marama toolset. We are also currently generalising MaramaTatau together with two other event handling specification approaches we have developed, ViTABaL-WS [14] and Kaitiaki [13], into a generic event handling framework. ViTABaL-WS provides a visual language for the design and construction of tool abstraction action-event-based architecture. Kaitiaki provides an extensible event-query-filter-action language for responding to propagated events. The generalisation of these three approaches within the Marama metatool framework will provide wider-ranging support for event-based system design and construction. One element of this is to extend the OCL expression language with higher-order function capabilities to provide enhanced expressability.

7. Summary

We have described an approach for constraint/dependency specification in a domain-specific visual language meta-tool. This borrows much from techniques used to support the spreadsheet metaphor, but in a situation with less concreteness. The innovation lies in combining well known technologies in the form of OCL and spreadsheet interfaces in a simple novel way drawing strength from both while mitigating their weaknesses. MaramaTatau augments the Marama meta-tool's meta-model designers, allowing tool developers to specify formulae over meta-models, combined with a one-way constraint system to compute values during tool usage. This allows for much simpler specification of dependency and constraint handling within Marama tools, compared to both the textual event handlers and Kaitiaki visual event handlers. The approach has some similarity to ClassSheets [5], but avoids the grid structure of that approach, and provides more mitigation of hidden dependencies. It considerably extends the visual metamodel annotation mechanism plus OCL expression of GME, providing many additional hidden dependency mitigations. Early developer feedback is very positive.

8. References

[1] A. Ledeczi., M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. In Proc. Workshop on Intelligent Signal Processing. 2001

[2] DSL Tools, http://msdn2.microsoft.com/en-us/vstudio/aa718368.aspx

[3] Eclipse EMF OCL plug-in, http://www.eclipse.org/modeling/mdt/downloads/?project=ocl

[4] EMF, http://www.eclipse.org/modeling/emf/

[5] Engels, G. and Erwig M., "ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications", *20th IEEE/ACM Int. Conf. on Automated Software Engineering*, 124-133, 2005

[6] GEF, http://www.eclipse.org/gef/

[7] Green, T. R. G. & Petre, M. (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J. Visual Languages and Computing, 7, 131-174

[8] GMF, http://www.eclipse.org/gmf

[9] 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

[10] Grundy, J.C., Hosking, J.G., Zhu, N. and Liu, N. Generating Domain-Specific Visual Language Editors from High-level Tool Specifications, *Proc. of 2006 IEEE/ACM ASE*, Tokyo, 24-28 Sept 2006, IEEE.

[11] Grundy, J.C., Mugridge, W.B., and Hosking, J.G. Static and Dynamic Visualisation of Software Architectures for Component-based Systems, *Proc. of the 10th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, June 18-20 1998, KSI Press, pp. 426-433

[12] Kelly, S., Lyytinen, K., and Rossi, M., MetaEdit+: A Fully configurable Multi-User and Multi-Tool CASE Environment, *Proc. of CAiSE'96*, LNCS 1080, 1996.

[13] Liu, N., Hosking, J.G. and Grundy, J.C. A Visual Language and Environment for Specifying Design Tool Event Handling, *Proc. of VL/HCC'2005*, Dallas, 2005.

[14] Liu, N., Grundy, J.C. and Hosking, J.G. A Visual Language and Environment for Composing Web Services, *Proc. of 2005 IEEE/ACM ASE*, Long Beach CA, Nov 7-11 2005.

[15] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A first-order visual language to explore the boundaries of the spreadsheet paradigm. Journal of Functional Programming, 11(2):155--206, March 2001

[16] Marama, http://www.cs.auckland.ac.nz/Nikau/marama/

[17] Mugridge, W.B., Hosking, J.G. and Grundy, J.C. Drag-throughs and attachment regions in BuildByWire, *Proc. of OZCHI'98*, Adelaide, Australia, Dec 1-4 1998, IEEE CS Press, pp. 320-327

[18] OCL, http://www.omg.org/docs/ptc/03-10-14.pdf

[19] Robbins, J.E. and Redmiles D.F. Software architecture critics in the Argo design environment, J Knowledge-Based Systems 11 (1998) 47-60.

[20] Zhu, N., Grundy, J.C., Hosking, J.G., Liu, N., Cao, S. and Mehra, A. Pounamu: a meta-tool for exploratory domain-specific visual language tool development, Journal of Systems and Software, Elsevier, *in press*.