# Generating Domain-Specific Visual Language Tools from Abstract Visual Specifications

John C. Grundy, *Member*, *IEEE*, John Hosking, *Member*, *IEEE*, Karen Na Li, Norhayati Mohd Ali, Jun Huh, and Richard Lei Li

**Abstract**—Domain-specific visual languages support high-level modeling for a wide range of application domains. However, building tools to support such languages is very challenging. We describe a set of key conceptual requirements for such tools and our approach to addressing these requirements, a set of visual language-based metatools. These support definition of metamodels, visual notations, views, modeling behaviors, design critics, and model transformations and provide a platform to realize target visual modeling tools. Extensions support collaborative work, human-centric tool interaction, and multiplatform deployment. We illustrate application of the metatoolset on tools developed with our approach. We describe tool developer and cognitive evaluations of our platform and our exemplar tools, and summarize key future research directions.

**Index Terms**—Metatool, domain-specific visual language, software tool, visual specification, model-driven engineering

## 1 Introduction

Software engineers use models to describe software requirements, design, processes, networks, tests, configurations, and code. Construction, engineering and computer systems professionals use models representing structures, plant, plumbing/electrics, materials, VHDL, electromagnetics, and processes/tasks. Health professionals have models for patient diagnoses, treatments, and imaging. Business, finance, and economics professionals use models to design and monitor processes/workflow. Families and friends may use models for family trees or to establish social networks. Our interest has been in the use of Domain-Specific Visual Languages (DSVLs) in these widely varied domains to assist domain users to better work with their complex domain models. A DSVL has a metamodel and visual notation allowing domain users to express complex models in one or more visual forms. Often, multiple visual forms are used to represent overlapping parts of the metamodel. Ideally, DSVLs afford a "closeness of fit" to the tool developer's problem domain [23], [65].

Working with models involves authoring, visualizing, navigating, transforming, understanding, managing, and evolving models. There is a demand for appropriate, usable, scalable, sharable, robust, and extensible tools to support these processes. Often multiple domain users must work together to author and review visual models. They sometimes want to model or access models in varying notations or interfaces, e.g., web or mobile device. They want effective tools to support the use of these DSVLs. However, building such tools is very challenging with the need for multiview, multinotational, and multi-user support, the ability for nonprogrammer tool developers to (re)configure specifications while in use, and an open architecture for tool extension and integration.

Current approaches to constructing DSVL tools suffer from a range of deficiencies. These include limited domain targets, the need to use complex APIs and code for developing even simple environments, and a complex edit-compile-run cycle for reflecting even minor changes. These deficiencies provide barriers to use and typically prevent domain users and even developers from producing suitable DSVL tools. Visual specification approaches, compared to writing custom code, have shown their advantages in minimizing design and implementation effort and improving understandability of programs [14], [17], [23], [27], [29]. This suggested to us that a visual language approach to support DSVL definition is likely to be a positive approach for the design and construction of domain-specific modeling environments, both for domain modeling tool users but also potentially for tool developers.

We focus on our conceptual/theoretic contributions to DSVL tool development and their realization in this paper. We describe the key motivation for this research (Section 2) and survey related work (Section 3). We then provide an

- *J.C. Grundy is with the Centre for Computing and Engineering Software Systems, Faculty of Information and Communication Technologies, Swinburne University of Technology, PO Box 28, John Street, Hawthorn, Victoria 3122, Australia. E-mail: jgrundy@swin.edu.au.*
- *J. Hosking is with the College of Engineering and Computer Science, Australian National University (ANU), RSISE Bldg, Canberra, ACT 0200, Australia. E-mail: john.hosking@anu.edu.au.*
- *K.N. Li is with SolNet Solutions Limited, Level 12, Solnet House 70 The Terrace, Wellington 6011, New Zealand. E-mail: Karen.Li@solnetsolutions.co.nz.*
- *N.M. Ali is with the Faculty of Computer Science and Information Technology, Universiti Putra Malaysia, Serdang 43400, Selangor, Malaysia. E-mail: hayati@fsktm.upm.edu.my.*
- *J. Huh is with the Centre for Software Innovation, Computer Science Department, and Auckland UniServices, The University of Auckland, Private Bag 92019, Auckland, New Zealand. E-mail: jhuh003@ec.auckland.ac.nz.*
- *R.L. Li is with Beef and Lamb New Zealand, Level 4, Wellington Chambers, 154 Featherston Street, Wellington 6011, New Zealand. E-mail: Richard.Li@beeflambnz.com.*

*Manuscript received 27 Feb. 2011; revised 1 Mar. 2012; accepted 28 Apr. 2012; published online 23 May 2012.*
*Recommended for acceptance by L. Osterweil.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-02-0059.*
*Digital Object Identifier no. 10.1109/TSE.2012.33.*

0098-5589/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

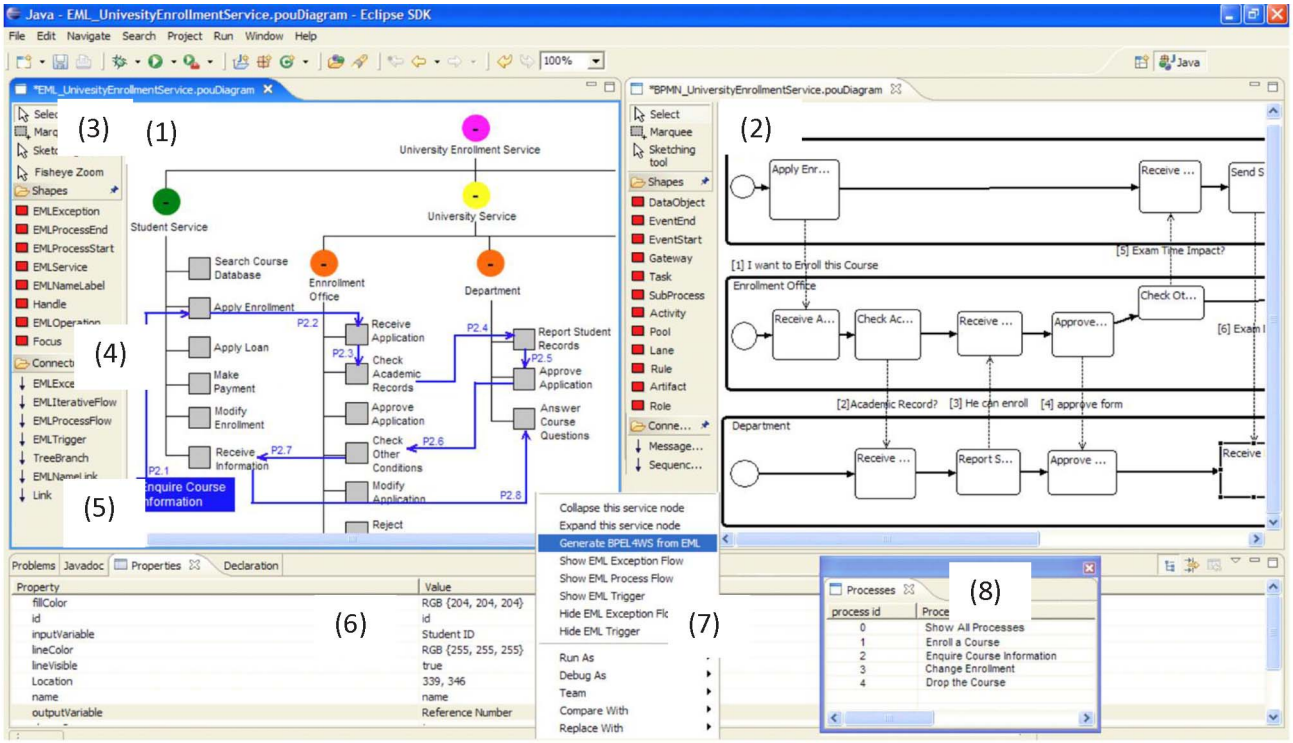



Fig. 1. MaramaEML diagrams: (a) tree-based services specification with process overlay, and (b) BPMN process flow.

overview of our metatool approach (Section 4) by describing its key features and conceptual foundation. We then describe our integrated set of meta-DSVLs with a focus on novel behavior specifications (Section 5), critic authoring, and model transformation (Section 6). These are followed by a set of human-centric elements supporting accessibility and collaboration (Section 7) and then platform realization (Section 8). We describe evaluation of our approach via exemplar DSVL tool development and a variety of more formal evaluations (Section 9), then provide discussion (Section 10) and conclude with a summary of key contributions from this research (Section 11).

## 2 Motivation

Software engineers use a range of models to describe software systems at various levels of abstraction. Some are very general and can be used to describe a wide range of software system characteristics. Some examples of such general-purpose visual modeling languages include the Unified Modeling Language (UML), Architectural Description Languages (ADLs), Entity-Relationship (ER) and Dataflow Diagrams (DFDs), and State Charts (SDs). This is akin to general purpose programming languages like C, C++, Java, etc., compared to domain-specific languages (DSLs) for special purpose domains, e.g., ATL for data transformation, BPEL for process orchestration, and the Ant build scripting language. Many tools have been developed to support these general-purpose modeling languages. As they are general-purpose, investment in bespoke visual modeling tools for these modeling languages has generally been worthwhile.

Domain-specific visual languages are more limited-domain modeling languages intended for modeling of limited classes of software systems or for modeling narrow aspects of systems, as DSLs are more limited, special-purpose forms of textual languages. One of the more successful DSVLs is the LabView visual language and environment [38], designed for instrumentation engineers configuring software and hardware interfaces in their domain. Other examples include various process modeling languages, e.g., Business Process Modeling Notation (BPMN) [75], load modeling languages, e.g., Form Charts [19], component and service composition [25], and visual modeling languages for specific software application domains, such as health care plans [43], business workflow [51], and statistical surveying [44]. As these have much more limited purpose and scope, large investment in developing sophisticated tools for such domains is sometimes hard to justify or afford.

Consider an environment to support business service and process modeling, such as our MaramaEML domain-specific visual language tool [51]. Two DSVL examples from this tool are shown in Fig. 1. MaramaEML needs to provide users with various visual notations for representing and understanding organizational services and process flows (Fig. 1(1)). In this example, the Enterprise Modeling Language (EML) DSVL is used in Fig. 1(1) to model a university student management application business processes. Other DSVLs may also be used, such as the Business Process Modeling Notation, an emerging standard for process modeling for both engineers and business analysis. Fig. 1(2) shows a BPMN process model for the "enroll student" business process. Such DSVL tools require sophisticated diagram editing tool features (Fig. 1(3)-(5)), detailed property editing (Fig. 1(6)), script and code generators, such as a BPMN to BPEL generator (Fig. 1(7)), and various analysis tools, such as consistency checking of

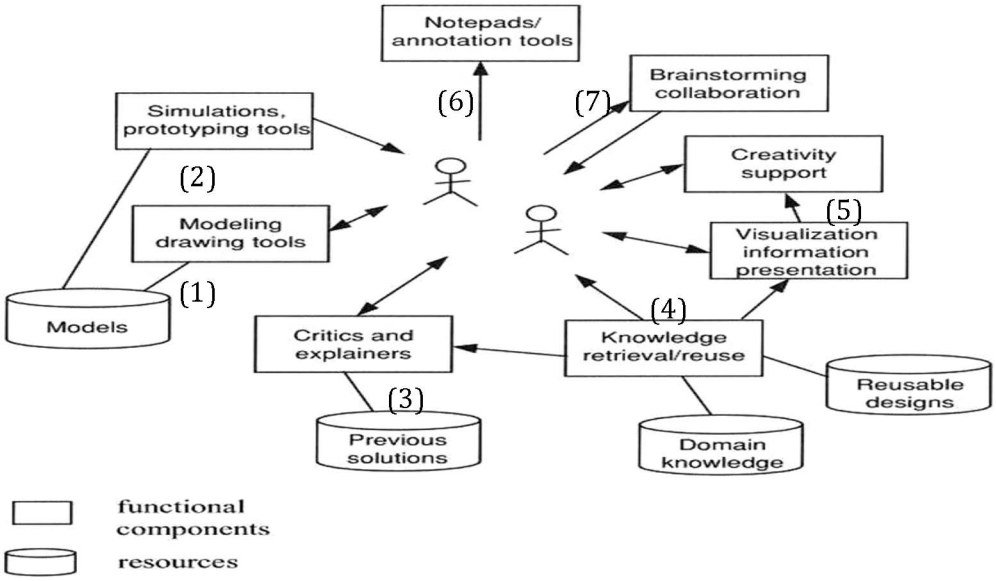



Fig. 2. Sutcliffe's design metadomain model, from [73].

process models (Fig. 1(8)). Developing such DSVL tool features in, e.g., Eclipse or Visual Studio is very time consuming, requires detailed knowledge of many platform APIs, requires significant coding and debugging, and they are difficult to maintain. As an example, to develop the EML tool in Eclipse using Eclipse Modeling Framework (EMF), OCL, and Graphical Editing Framework (GEF) projects to produce only a basic visual editor, approximately 2,500 lines of code need to be produced (ignoring the autogenerated EMF data structure code), roughly 2,100 implementing the GEF graphical editor and 400 implementing various metamodel constraints and behavior. Use of Graphical Modeling Framework (GMF) supports generation of around 1,800 lines of the graphical editor and additional controller code. The tool developer still needs to intimately understand and use Eclipse GEF, GMF, EMF, and OCL APIs, along with the different Event notification and Command frameworks, the XMI serializer, the Eclipse plug-in and parts models, the OSGi-based plug-in configuration and deployment tool, and complex interdependencies between all of these.

As developing such DSVL environments is such a complex task, it is generally exclusive to experienced software developers. Ideally, we want this process to be simplified by leveraging metatool capabilities. Given many DSVLs may be useful and used by nontechnical Tool Developers, ideally we want nonprogrammer tool developers to be able to develop visual notations and tools of relevance to their domain using their own domain knowledge. Thus, our key goals include: 1) making DSVL tool implementation easier for experienced domain modelers (who may not always be experienced software developers), and users familiar with basic modeling concepts, e.g., Extended Entity Relationship (EER), OCL, and metamodels; 2) allowing users to construct basic DSVL tools within one day, plus time for additional complexity such as back-end code generators; and 3) leveraging the strength of the Eclipse platform, as our previous efforts with our standalone Pounamu [79] left us with infrastructure support needs that were too large and an inability to integrate seamlessly with other work.

Sutcliffe provides a useful conceptual framework for design modeling tools [73], shown in Fig. 2. He argues that tools for model management should provide suitable model authoring (drawing) tools (1); simulation and prototyping support (2); design critics (3), knowledge reuse (4), and visualization (5) support; and annotation (6) and collaboration support (7).

Using Sutcliffe's conceptual model and our own experiences developing previous metatools, we have identified some key requirements for a metatool to construct DSVL tools as per our goals established above:

- Specifying modeling elements (Requirement 1)—These comprise
  - metamodel(s) representing canonical model(s) of domain-specific information, including entities and relationships, in the target DSVL;
  - visual elements, made up of various icons and connectors, which form the visual representation(s) of domain models and elements; and
  - diagrammatic views, which comprise notational elements for each DSVL diagram type and view-to-model mappings for the management of view-model consistency.
- Specifying modeling behaviors (Requirement 2)—These include dynamic and interactive tool effects such as event and constraint handling for both model and view manipulations and automated operations or processes.
- Model critiquing and transformation (Requirement 3)—To support proactive feedback on model quality and the exchange of view and model information with other tools, and back-end code generation.
- Human-centric modeling (Requirement 4)—Including scalable, sharable, usable, and intelligent support for collaborative and sketch-based editing and review.

- Modeling platforms (Requirement 5)—leveraging existing IDE facilities and related tools and making models available to domain users in appropriate ways.

## 3 Related Work

Three main approaches exist for the development of visual, multiple view DSVL environments: reusable class frameworks, diagram generation toolkits, and metatools.

General-purpose graphical frameworks provide low-level yet powerful sets of reusable facilities for building diagramming tools or applications. These include MVC [45], Unidraw [74], COAST [72], HotDoc [13], and Eclipse's GEF [1]. While flexible and powerful, these frameworks typically lack abstractions specific to multinotation and multiview visual language environments. Thus, construction of DSVL tools is very time consuming. For example, supporting multiple views of a shared model in GEF requires a significant programming effort. Given the challenge, a variety of special purpose frameworks for building multiview diagramming tools have been developed. These include JViews [30], IBM ILOG JViewsDiagrammer [3], and NetBeans Visual Library [6]. These offer reusable facilities for visual language-based environments, but still require detailed programming and an edit-compile-run cycle, limiting their ease of use for exploratory development and for tool developers without detailed technical knowledge.

A number of more targeted diagram generation toolkits have been produced to make DSVL tool development easier. These include Vampire [60], VisPro [77], JComposer [30], PROGRES [70], DiaGen [62], VisualDiaGen [63], Merlin [4], and VEGGIE [78]. All of these diagram generation toolkits use code generation from high-level specifications. Some use metamodels and associated editor characterization as their source specifications, e.g., JComposer and Merlin (using EMF model). Others, e.g., DiaGen, VisualDiaGen, PROGRES, VEGGIE, and VisPro, use formalisms such as graph grammars and graph rewriting for high-level syntactic and semantic specification of visual tools. Code generation approaches suffer from similar problems to many toolkits: often requiring an edit-compile-run cycle and difficulty in integrating third-party solutions. Tool developers sometimes have to resort to code for some editor capabilities (e.g., customization of shapes in Merlin), or cannot add some desired support features to their target tools. Formalism-based visual language toolkits typically limit the range of visual languages supported and are often difficult to extend in unplanned ways. They also require understanding of the formalism, most commonly graph grammars that specify a set of valid transformations from one graph state to another. Historically, graph-grammar-based visual editors had more limited editing capabilities than toolkit or framework-coded tools, requiring valid graph transforms only and menu-driven editing approaches. More recent toolkits such as DiaGen provide graphical editors with growing flexibility.

A third approach to realizing DSVL tools is using metatools. Metatools provide a purpose-designed IDE for generation, configuration, and exploratory development of other tools. These include KOGGE [20], MetaEdit+ [41], MOOT [67], GME [47], Pounamu [79], TIGER [21], DiaMeta [64], Eclipse Graphical Modeling Framework [2], $\mathrm{AToM}^3$ [46], and Microsoft DSL Tools [5]. Metatools for DSVL environments typically provide separate specifications of different tool aspects. High-level definitions of tool metamodels, shapes, connectors, and mappings from metamodel elements to shapes and connectors are used to effectively generate a tool structure implementation. While a popular mechanism has been provided to allow DSVL tools to be quickly generated and configured with minimal user specification effort, features such as complex editing behavior, tool integration, code generation, design critics, human-centric editing, and accessibility are generally not directly supported and must be hand coded if required. Typical distinctions of the existing metatools include the paradigms used for metamodeling (e.g., UML used by DSL Tools, EMF used by GMF, ER used by Pounamu, and graph grammars used by TIGER), the variety of facilities offered for visual notation design (e.g., drawings, forms, templates, and abstract specifications), the directness of multiple view specification support (e.g., automated, via visual mapping, or with the need for substantial coding), and, of particular concern to us, the abstraction level of DSVL tool behavioral specification support (e.g., using scripting languages, building blocks provided in the metatool frameworks or visual notations). Microsoft DSL Tools provides an integrated visual specification editor with designated parts for metamodel, visual notations, and their representational mappings. Multiple, linked views are, however, not directly supported and realizing them in generated tools requires much coding and configuration. Model validation, diagramming rule definition, and artifact generation are well supported in the SDK through the use of framework APIs, metadata attributes (annotations), object-oriented inheritance, and template-based generators. Reuse and integration are also well-supported, leveraging Visual Studio's extensibility. However, these are not raised to a visual abstraction level. Pounamu provides built-in support and a form-based mapper for multiple, linked views. It also allows some limited form of visual event handling behavior specification. DiaMeta allows rule-based diagram layout control behavior to be specified using metamodels in EMF. $\mathrm{AToM}^3$ provides a DSVL (called SLAMMER [31]) for the use of generalized visual patterns for DSVL model measurement and redesign. MetaEdit+ provides common rules for Tool Developers to choose/adapt, and automatically delivers them in model instances. For code generation and integration, advanced built-in scripting commands are used. GMF supports live validation of diagrams. It also allows models to be specified in OCL, leveraging the EMF validation framework. C-SAW [76] in GME supports model constraint specification in OCL using a separate form-based editor view. Most metatools aim for a degree of round-trip engineering of the target tools. Typically, they provide support for their target domain environments and tool developers may need little technical programming ability to use the metatools. However, many are limited in their flexibility in terms of target tool capabilities able to be specified and their integration with other tools. Pounamu and IPSEN are good examples of these limitations, and

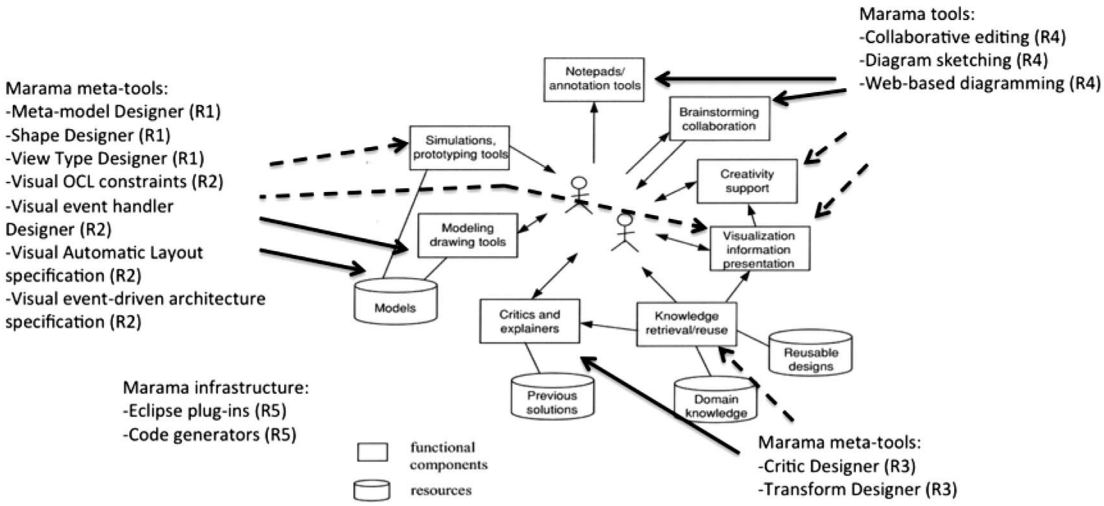



Fig. 3. Relating Sutcliffe's model to Marama capabilities.

additionally incur large effort to build features provided by general-purpose IDEs. Thus, a recent trend has been to build both diagram editor toolkits and metatools on top of existing IDES such as Eclipse and Visual Studio. These can then leverage save/load resource management features, modeling frameworks, graphical and text editor frameworks, a common user interface look and feel, and many third-party extensions.

Considerable research has gone into providing proactive critiquing support in software IDEs, DSVL tools, and other design-oriented domains. These give proactive support to users as they model. Good examples include design advice in ArgoUML and advice on Java programming constructs in Java Critiquer [69], [71]. However, no high-level critic definition approaches have been incorporated into DSVL metatools. Model transformation and code generation support has been recognized as a critical feature for many DSVL tools employed for model-driven engineering problems. This has included approaches such as enterprise data mapping, GXL, and VMTS [11], [34], [48]. Typically, these approaches have been standalone model transformation or exchange tools rather than integrated DSVL metatool support features. Providing more human-centric editing capabilities for DSVL tools has also been recognized in much recent research. A number of approaches to providing sketching-based support have been developed, such as extensions to DiaGen for sketch-based recognition [12]. Diagram differencing, merging, and collaborative editing support have been explored by Mehra et al. [61] and Lin et al. [52]. Most of these approaches provide fixed, closed groupware functionality, however, and we desired more flexibility over target visual design tool collaborative work facilities. In addition, we wanted these capabilities to support integration with existing tools. Thin-client, or rich Internet application (RIAs) interfaces, have been explored; examples include MILOS [59], a web-based process management tool, BSCW [9], a shared workspace system, WebCASRE [56], which provides software reliability management support, web-based tool integration, and CHIME [39], which provides a hypermedia environment for software engineering. Most of these provide conventional, form-based web interfaces and lack web-based diagramming tools. Recent efforts at building web-based diagramming tools include Seek [42], a UML sequence diagramming tool, NutCASE [22], a UML class-diagramming tool, and Cliki [57], a thin-client metadiagramming tool. All have used custom approaches to realize thin-client diagramming. They also provide limited tool tailorability by tool developers and limited integration support with other software tools. Many are standalone efforts, whereas ideally metatools should be able to support these capabilities for all target DSVL tools.

In summary, a number of approaches have demonstrated the capability to capture domain model elements using high-level specifications (our Requirement 1). However, the majority require detailed programming and framework knowledge or understanding of complex formal information representation models (e.g., graph grammars) for DSVL tool development. Few have managed to support behavior specifications accessible to nonprogrammer tool developers (Requirement 2). There is to date limited support for DSVL tool critic editing based on knowledge (best practices) reuse and for model transformations (Requirement 3). Few approaches support simple, live, evolutionary, and collaborative development of DSVL tools with good accessibility and the use of Tool Developers' own domain knowledge (Requirement 4). More approaches are leveraging IDEs to realize target DSVL tools, but these still lack high-level support for tool integration (Requirement 5).

## 4 Overview of Our Approach

We wanted to simplify the DSVL tool development process by extending metatool capabilities. Our approach is to generate DSVL tools from a variety of high-level, visual specifications in a metatool, called Marama. Fig. 3 relates our key requirements for DSVL metatools from Section 2 above to Sutcliffe's Design metadomain model [73].

Our Marama approach addresses most of the components in the Design metadomain, providing strong support in some areas and partial in others. Our core approach is a set of visual, declarative specifications of domain metamodels, their visual representations, and their views. These include an extended entity-relationship modeler

for the specification of domain specific metamodels (defined using a "Metamodel Designer"), a WYSIWYG "compose-and-edit" approach to visual notational element specification (the "Shape Designer"), and a mapping tool for the filtering of model elements into (possibly multiple) views (diagrams) and the facilitation of consistent view and model editing (the "View Type Designer"). These collectively address our key Requirement 1.

Several approaches are used to specify advanced DSVL tool behaviors, our key Requirement 2. One is a visual, declarative specification of constraints on models that augments the Metamodel Designer. A form of OCL constraints is supported by visual representations of model element dependency and a spreadsheet-like model/formulae metaphor. A visual, imperative specification of event-based behavior provides a set of "Event Handler Designers." A declarative augmentation of shape designs and view type designs is used to specify (limited) automatic layout functionality. A high-level architectural component model allows tool developers to specify intra and intertool communication and coordination. These all make use of an integrated event handling behavioral model based on event-condition-action rules. Together, these metatools provide strong support for defining models, visual notations for models, and drawing tools for authoring visual notations of models. They provide basic support for information visualization, visual debugging, and simulations in target DSVL tools.

A visual, template-based critique and feedback authoring system is used to specify a set of "design critics" for proactive advice to DSVL tool users (the "Critic Designer"). A visual, tree-based schema mapping approach facilitates model transformation, model import, and code generation (the "Transform Designer"). Together these support a range of proactive design critics for a target DSVL tool and a range of information exchange, reuse, and code generation facilities, addressing our key Requirement 3.

We have experimented with a range of ways of supporting human-centric modeling in Marama DSVL tools. These include a sketched-based overlay for diagram editing via intelligent ink annotations, support for multi-user collaborative editing, and thin-client, web-based interfaces for more accessible information presentation and authoring. Together these allow generated DSVL tools to support flexible annotation, collaboration, and brainstorming. They provide some limited support for creative design and information visualization and thus partially address our Requirement 4. We have realized Marama as a set of plug-ins using the Eclipse IDE, addressing our Requirement 5.

## 5 Metamodel and Behavioral Specifications

We use our MaramaEML business process modeling tool introduced in Section 2 as a running example of a complex DSVL tool to be specified and generated with Marama. Consider a tool developer wanting to develop such a tool to enable high-level business process modeling, BPEL script generation, integration of a third-party LTSA model checker for BPEL, and information visualization support. We first show how the basics of such a DSVL tool can be specified in Marama. Later, we illustrate how we can augment the basic tool specification with design critics, model transformation, and human-centric modeling support. We then describe our realization of the Marama DVSL metatool using Eclipse and Microsoft DSL Tools IDEs.

### 5.1 Specifying DSVL Structural Aspects (Requirement 1)

Structural elements, including entities, relationships, shapes, connectors, and view types, form the backbone of a DSVL tool specification. Defining domain specific concepts is a mind-mapping process of abstracting out entities, relationships, subtyping, roles, attributes, and keys as encountered in the tool developer domain. In Marama, this process is metamodeling. The Marama Metamodel Designer tool, illustrated in Fig. 4, uses an EER representation. We chose an EER approach, rather than MOF or UML, for simplicity for our target tool developer community, which includes nontechnical DSVL tool developers. The Marama EER metamodel can, however, be transformed to and from other metamodeling representations. Fig. 4 shows a basic metamodel for representing MaramaEML BPMN constructs.

In this example, the MaramaEML tool designer has specified a range of entities (green square icons) to represent fundamental domain concepts, e.g., Activity, StartEvent, StopEvent, Gateway, Comment, Swimlane, etc. Some of these have been generalized to supertypes, e.g., Element, Event, and ProcessElement, via generalization relationships (unfilled arrow pointing to general type). This allows types to share common information, including attributes and associations. Several associations have been specified, shown as pink oval rectangles, e.g., Comment-to-Element, Activity-to-Activity, etc. These connect elements via named association links. Multiple metamodel diagrams are possible to manage complexity.

The tool developer can also specify "event handlers" which detect editing events or provide tool users with pop-up menus to invoke additional functionality. In this example, a GenerateUniqueID event handler creates a unique ID upon the creation of new elements. Event handlers are tool developer written Java-coded scripts, reused, and parameterized scripts, or are generated by various other Marama visual specification metatools (see later). Marama provides a comprehensive set of APIs, allowing tool developers to query and manipulate any aspect of the Marama metamodel or diagram data structures. Event handler code can also access any Eclipse APIs used by Marama.

A WYSIWYG Shape Designer tool, illustrated in Fig. 5, allows rapid composition of icons and connectors for representing the domain-specific concepts defined in the metamodel. Icons and connectors are specified in a generic, abstract form via drag-and-drop. This uses a semiconcrete representation of the specified shape and connector notation for immediate design feedback. In this example, Start and Stop event shapes (ovals), an Activity shape, a Group shape, a Comment shape, and a Gateway shape have been designed. Some of the editable attribute values for the Activity shape specification are shown. Some associations are shown, including event flow, comment anchor, and grouping.

The View Type Designer tool, illustrated in Fig. 6, specifies which visual elements are included in a diagram (which we call a "view type") and their relationships to the

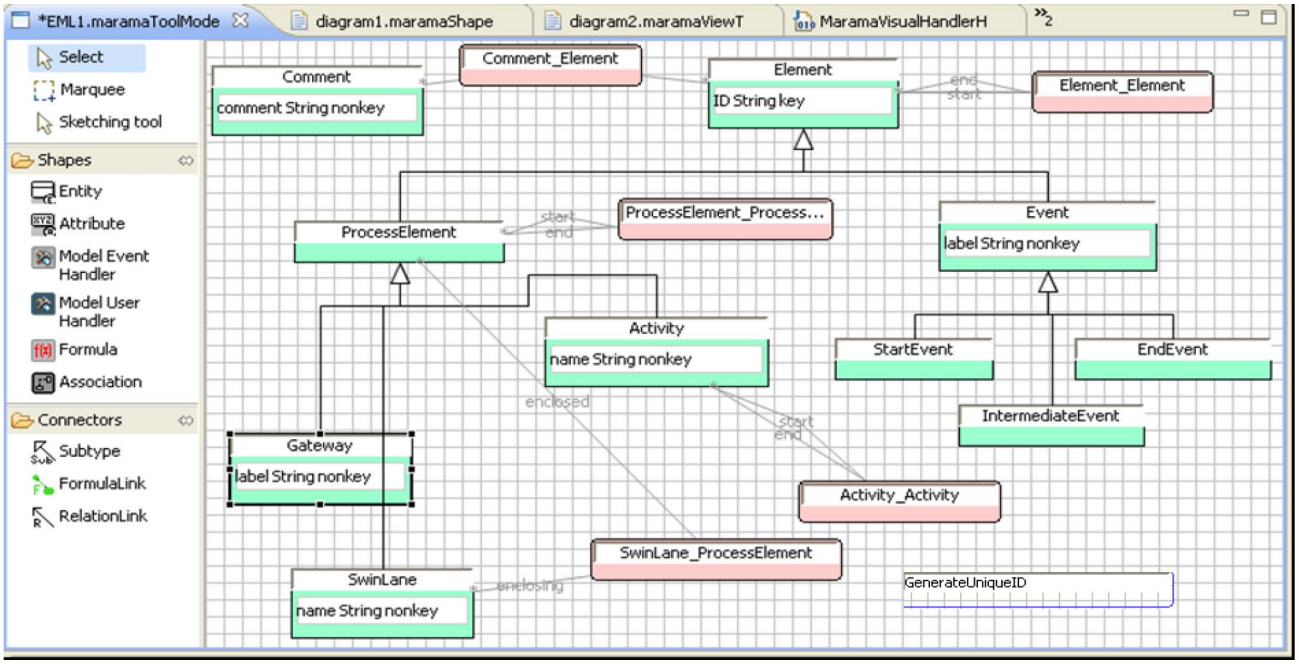



Fig. 4. An example MaramaEML metamodel in the Marama metamodel designer tool.

underlying model elements (including attribute mappings). In this example, the main BPMN view type is specified. This includes Activity, Group, Comment, Event, and gateway shape mappings to appropriate metamodel entities, and mappings of various connectors such as Flow and Comment anchored to metamodel associations. Various attributes of shapes and connectors can be mapped to metamodel entity and association attributes. This implies automatic maintenance of a bidirectional consistency between realized view and model element attributes in models developed using the generated DSVL tool. A view type wizard is provided to select defined entity, association, shape, and connectors and generate an initial view type. View types may also have event handlers defined. For example, in this example, GroupContainsProcess and AlignContainedShapes are "event triggered" handlers for managing grouped BPMN process elements inside a Group shape. A generateBPEL event handler is a user-selected pop-up menu item that generates BPEL script from the BPMN metamodel elements.

Having specified a DSVL or set of DSVLs using these basic structural specifications (metamodel representations of domain concepts, their visual representations, and view mappings), users can, with no further work, realize the specified tool and use it to create domain specific models and diagrams using the DSVL(s). A model project contains one model instance with multiple view (diagram) instances, all

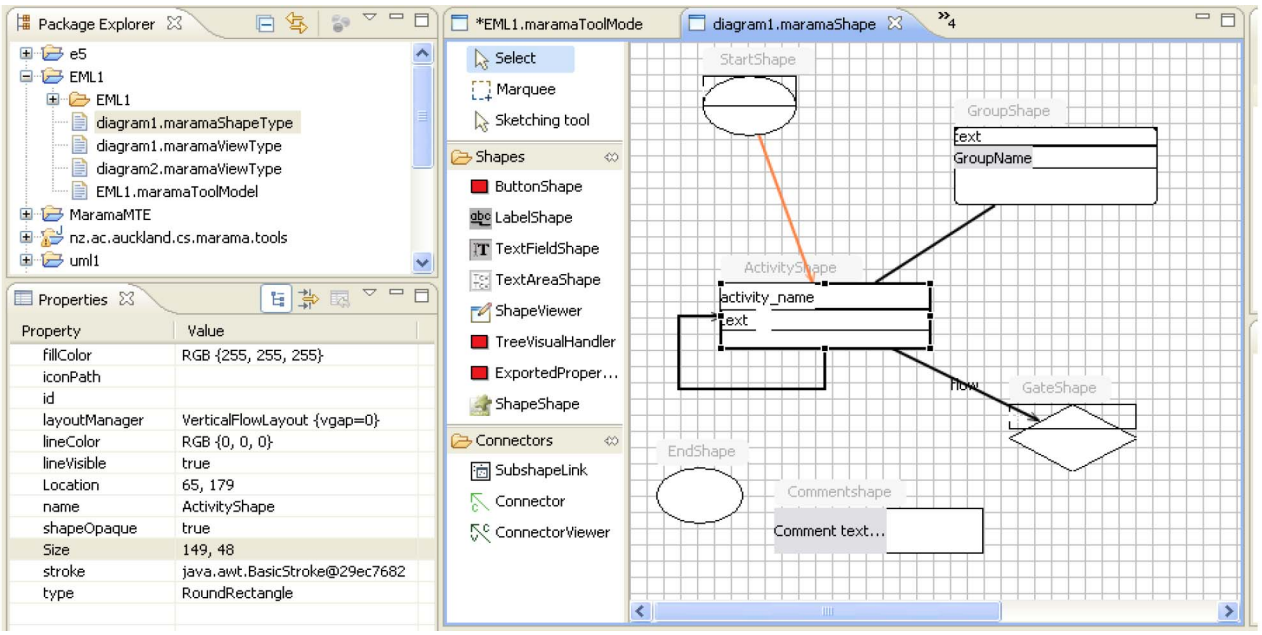



Fig. 5. An example of some MaramaEML BPMN concrete notation shape and connector designs in the Marama shape designer.

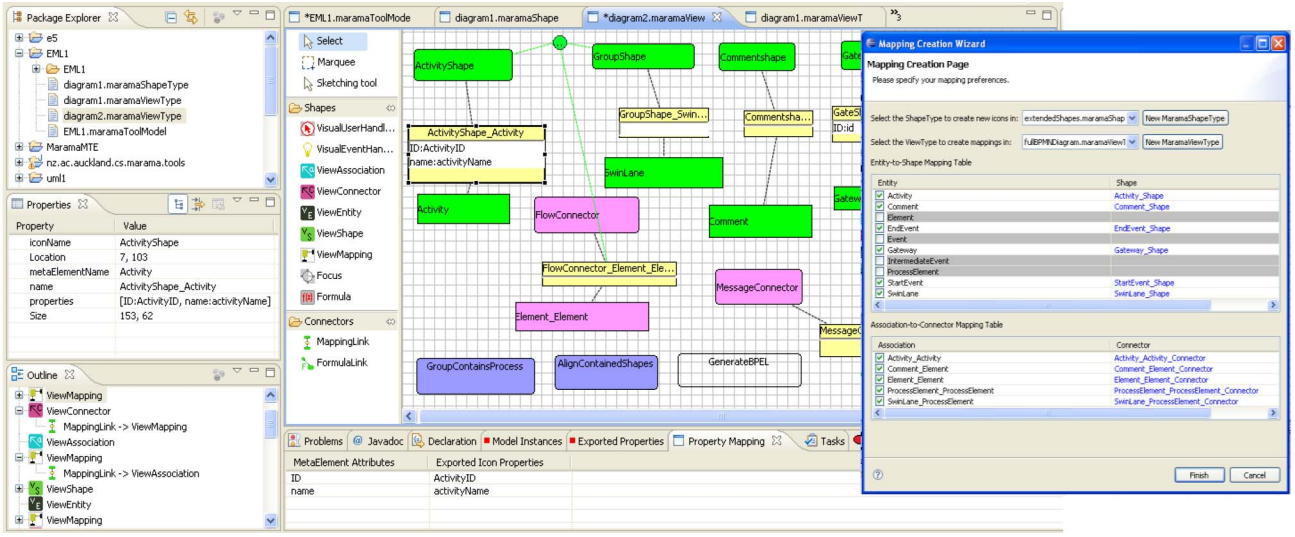



Fig. 6. Example of Marama view type designer (left) and view type creation wizard (right).

kept consistent with one another. Both the model and view instances can be manipulated via user interactions. A view instance realizes all the icon and connector types defined in the View Designer as Palette tools. As per the mappings of visual elements (icons and connectors) and model elements (entities and relationships) specified in the View Designer, an icon/connector instance in the view instance automatically generates a model entity/relationship, with appropriately mapped property values. Unmapped visual/model properties of a visual/model element are persisted independently. The model project contains a model file (file extension ".model"), which stores the runtime model state.

## 5.2 Specifying DSVL Tool Behavioral Aspects (Requirement 2)

Developing modeling behaviors is a key challenge in DSVL tools. Many approaches require programming knowledge and access to complex APIs. Our Marama event handlers written in Java code are very powerful but suffer from these same problems. However, behavioral specification from a high-level abstraction is generally difficult to achieve. Appropriately chosen metaphors are important for mapping a specification onto a user's domain knowledge, not only for structure but also for behavior specifications.

In Marama, we chose a declarative spreadsheet-like metaphor to specify model-level dependencies and constraints and an imperative dataflow-like Event-Condition-Action-based metaphor for view-level event handlers. We use a subscribe-notify Tool Abstraction (TA) metaphor to describe event-based tool architecture and multiview dependency and consistency. These three different metaphors were generalized from our earlier work on domain-specific event handling specification [53], [54], [55], integrated via a common model and unified user interface representation [50].

### *5.2.1 Declarative Model Constraint Specification*

A number of approaches separate model constraint specifications from their metamodel specifications [5], [41]. This causes potentially serious *hidden dependency* issues (one of the Cognitive Dimensions (CD) framework dimensions [23]). We believe constraints such as attribute value boundaries and dependencies, relationship multiplicities, and cyclic reference checking can be presented more simply and clearly within the same metamodel specification by annotating existing contextual elements. MaramaTatau is an extension of Marama's EER metamodel designer specifications adding declarative dependency/constraint specifications and high-level visual annotations. Value dependencies and modeling constraints are state-change events handled in Marama via a unidirectional change-propagation mechanism with side-effects to dependent components, the same approach used by formula evaluation in spreadsheets. However, we wished to minimize Cognitive Dimensions [23] tradeoffs such as hidden dependency and visibility issues between constraint and metamodel specifications that are common in spreadsheet-like approaches. MaramaTatau supports visual construction of formulas to specify model structural dependencies and constraints at a type rather than the usual spreadsheet instance level. We chose OCL as the primary textual formula notation as OCL expressions are relatively compact, OCL has primitives for common constraint expression needs, OCL is a standardized language, and the quality of OCL implementation is increasing.

In Fig. 7, we have used MaramaTatau to extend the MaramaEML metamodel with a constraint specifying that a StartEvent must have at least one Activity connected to it. Small green circular annotations on an attribute or entity indicate that an OCL formula has been defined to calculate a value (e.g., the id of an Element) or provide an invariant (e.g., the cardinality constraint on a StartState). We use green arrowed lines to show formula dependency annotations and gray borders to annotate sensible elements to be involved in a formula construction at a certain stage. Formula construction can be done textually, via a Formula Construction (OCL) view, or "visually" by direct manipulation of the metamodel and OCL views to automatically construct entity, path, and attribute references and function calls. Clicking on an attribute places an appropriate reference to it into the formula.

In this example, the tool developers create a formula for the StartEvent entity (1). They then begin specifying the constraint using a combination of the visual elements in the

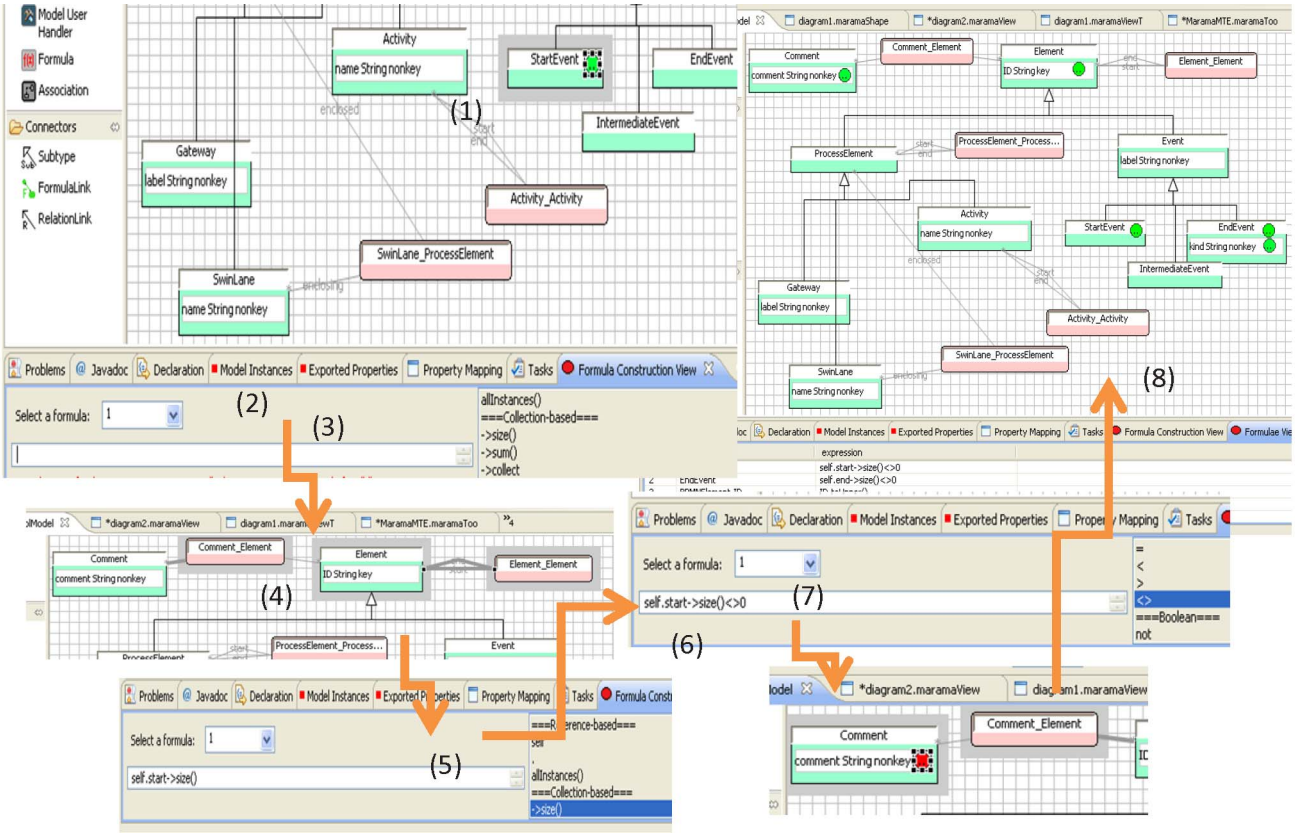



Fig. 7. Examples of visual constraint specification using the MaramaTatau extensions to the Marama metamodel designer.

metamodel and the formula editor view (2). They specify the StartEvent entity (self) by clicking on it (3), which highlights in the visual metamodel elements, associations, and/or attributes that can next be validly added to the formula (4). In this example, the developers choose its named association "start," linking StartEntity to the first Activity entity for the BPMN specification (5). Clicking on a relationship and then an attribute generates a path reference in the formula (self.start in this example). A difference from the spreadsheet approach is that only certain elements are semantically sensible at each stage of editing, whereas in spreadsheets, almost any cell may be referenced. The tool developers then specify the size() function, from the available functions list on the right (6), and add a constraint of <>0 (i.e., a StartEntity must have one or more connected Activity entities to be valid).

The cardinality constraint on the Service entity is thus specified by the OCL expression"self.start->size()<>0." When this formula evaluates false for a StartState in a model instance, a constraint violation error is generated, with a problem marker representation appearing in the Eclipse Problems view to provide the user with details of the violated constraint. In this example, to solve the error the user needs to connect the StartState entity to an Activity entity in the BPMN diagram. When this is done, the constraint evaluates to true and the constraint error is removed from the Problems view.

We have carefully defined interaction between the OCL and metamodel views to enhance visibility and mitigate hidden dependency issues. OCL and EER editors are juxtaposed, improving visibility, and simple annotation of the model elements indicates formulas related to them are present and semantically correct/incorrect. Formulas can be selected from either view so constraints can be readily navigated to/accessed. The dependency links permit more detailed understanding of a formula. The annotations are modified dynamically during editing for consistency. Dependencies are only made visible if a constraint is selected to minimize scalability issues and support task focus. The approach is similar to conventional spreadsheet dependency links, but applied to graphical modeling. Constraints with formula errors are highlighted in red (7). A set of example constraint indicators and formulas are shown in the final example screen dump (8).

### 5.2.2 *Declarative Diagram Constraint Specification*

Some specialized visual relationships in views, such as various composite icon layout mechanisms, can also be expressed declaratively using the same formulaic approach. Instead of using Java event handler specifications, the tool developer can reuse some limited visual specifications. We have extended our OCL-based metamodel dependency/constraint specification technique by adding some reusable functions and applying them to iconic notations at the view level. Fig. 8a shows a visual constraint specified in the BPMN view type design of MaramaEML. The enclosure relationship (icon is contained in but can be moved within the enclosing icon) is specified between a GroupShape and ActivityShape, using the FlowConnector connector to manage this relationship, i.e., all AcitvityShapes related to a GroupShape by a FlowConnector are "grouped" with that GroupShape. When the GroupShape is moved or resized, Marama automatically moves and resizes the grouped ActivityShapes, Fig. 8b. Adding, moving, resizing, or

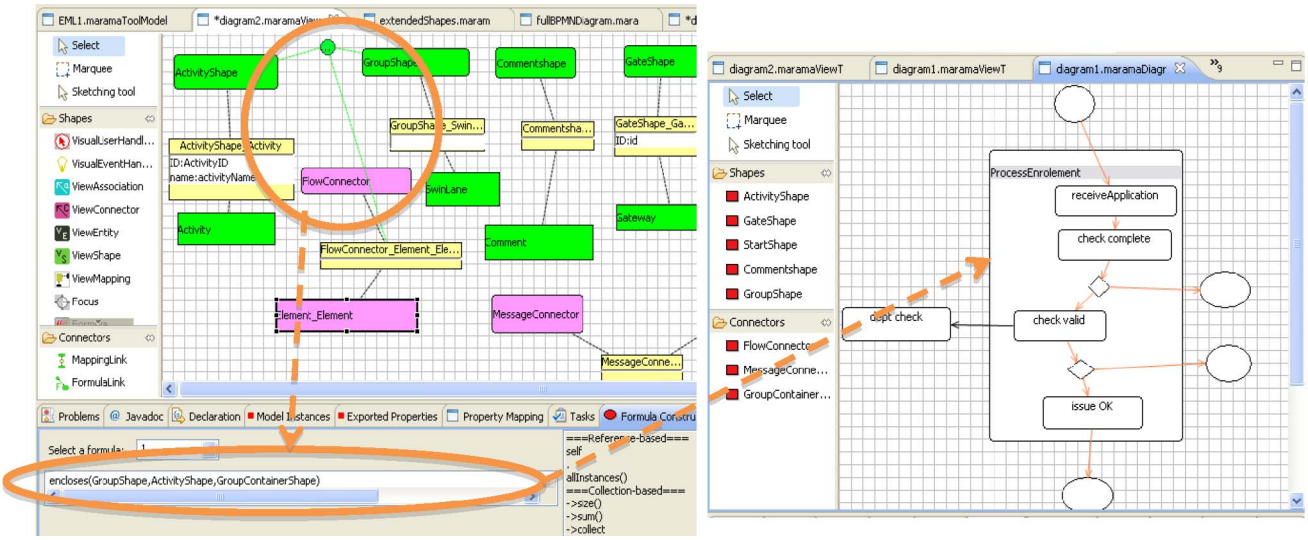



Fig. 8. (a) Example of an encloses() shape layout constraint being specified in the Marama view type designer and (b) its effect in the Marama generated BPMN editor.

deleting ActivityShapes in the group may cause the GroupShape to be automatically resized/moved to continue to contain them. Other constraints that can be specified in this way include full containment with vertical or horizontal layout of contained shapes, shapes autoaligned ("pinned") to the edge of another shape, and shapes autolocated within another shape.

### 5.2.3 Declarative Diagram Layout Specification

More complicated autolayout using trees and force-directed layout is supported by further visual augmentation of Shape and view type specifications. MaramaALM (Automatic Layout Mechanism) is an extension to the shape and view type designers that allows shapes to be specified as participating in complex tree (vertical or horizontal) and/or automatic force-directed layouts [66]. Fig. 9 left shows the tool designer augmenting shape designs in the Marama shape designer with visual annotations (small dark green octagon shapes). These indicate participation of the shape in automatic layouts in target diagrams. The augmented view type designer in Fig. 9 right shows layout event handlers added to a view type that will use these augmented shape type specifications to apply tree and/or force-directed layouts in the target Marama design tool. The tool designer specifies the shapes that participate in the tree or force-directed layout, connectors used to link these related shapes and configurations, e.g., horizontal or vertical tree, autoresize or not, and amounts to space and autorelocate.

Fig. 10 shows the specified autolayouts in use. In the top figure, a diagram has hierarchical tree layout. The user can change this to a horizontal layout as shown on the right. Marama automatically relays out the diagram components for shapes participating in the tree. Fig. 10 (bottom) shows a force-directed layout in use. The right-hand side shows the resizing and repositioning of shapes in the view that participate in the force-directed automatic layout specified above. Note MaramaALM supports a combination of tree and force-direct layouts for the same diagram, e.g., a horizontal tree lays out the oval shapes and a force-directed layout lays out the rounded rectangle shapes.

### 5.2.4 Visual Event Handler Specification

Our declarative spreadsheet-like approach allows tool designers to specify constraints in a simple visual and declarative way. However, there are limitations with this approach. It is awkward to express more operational behaviors such as composite queries, filters, and actions in event handling, and the flow of data between them. To

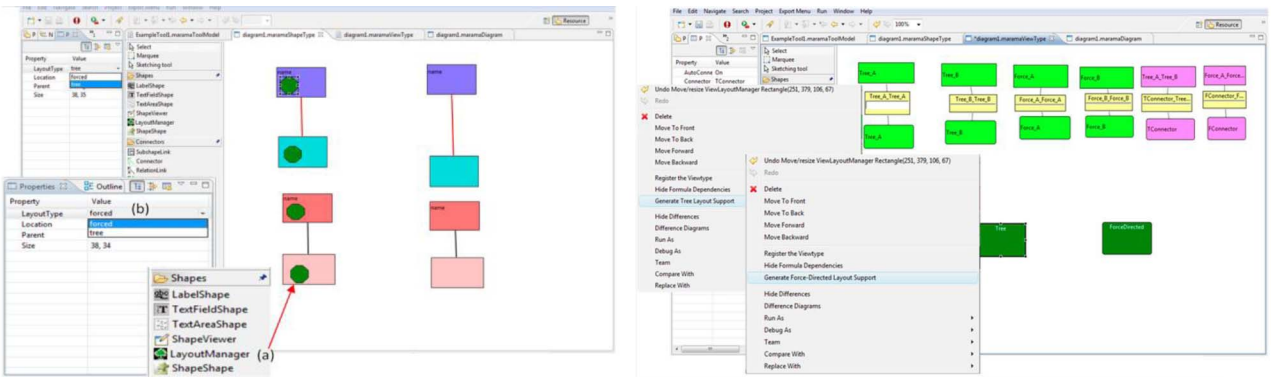



Fig. 9. Specifying participation in autolayouts (left) and type of autolayout(s) to add to a view type definition (right).

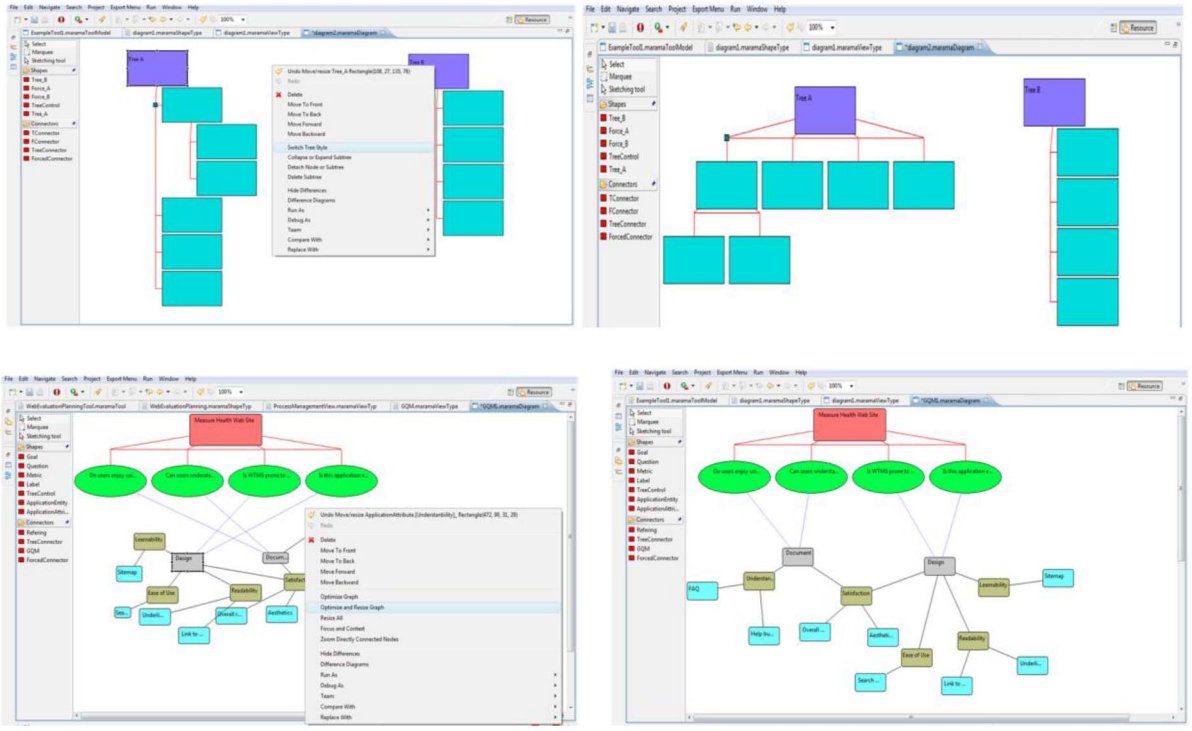


Fig. 10. Tree layout (top) and force-directed layout and optimization (bottom).

address this, we developed a visual event flow language (Kaitiaki): an "Event-Query-Filter-Action (EQFA)" notation for expressing view-level constraints/operations. The approach is based on our earlier Serendipity [27] visual event processing language. When constructing event handlers, tool developers select an event type of interest, add queries on the event and Marama tool state (usually diagram content or model objects that triggered the event), specify conditional or iterative filtering of the event/tool state data, and then state-changing actions to perform on target tool state objects. Complex event handlers can be built up in parts, via subviews, and queries, filters, and actions can be parameterized and reused.

The visual language design focuses on modularity and explicit representation of data propagation. We have avoided abstract control structures and used a dataflow paradigm to reduce cognitive load. Key visual constructs are events, filters, tool objects, queries on a tool's state, state changing actions plus iteration over collections of objects, and dataflow input and output ports and connectors. A single event or a set of events is the starting point. From this data flows out: event type, affected object(s), property values changed, etc. Queries, filters, and actions have parameter bindings via data propagated through inputs. Queries retrieve elements from a model repository and output data elements; filters apply pattern matching to input data, passing matching data on as outputs; actions execute operations which may modify incoming data, display information, or generate new events.

Queries and actions execute when input data are available (data push). If there are no input parameters, queries and actions trigger whenever parameters to a subsequent flow element have values (pull). We predefined a set of primitives for these constructs, providing operations useful for diagram manipulation by abstracting from a large set of diagram manipulation examples. These involve collecting, filtering, locating, or creating elements, property setting, relocating/alignment, and connection. Multiple flows are supported. Concrete DSVL icons, specified in the shape designer, are also incorporated into the visual specification of event handling as placeholders for Marama tool state to annotate and mitigate the abstraction, making the language more readable.

Fig. 11 shows an event handler specified for aligning ActivityShape shapes. The handler responds to a Marama "shapeAdded" or "ShapeMoved" event (1). The concrete representations of Activity and Gateway shapes (2) filter the added and moved events to only these types of shapes. A further filter checks that the Activity and Gateway shapes are contained within a Group shape (3). All of the shapes within the GroupShape are then retrieved by a Query (4) and an Action aligns all of the grouped shapes (5).

#### 5.2.5 *Visual Event-Based Architecture Specification*

The declarative and imperative constraint/event handling approaches described above are of low-to-medium abstraction and lack the ability to describe and affect the overall high-level architecture of a DSVL tool. We chose to use the Tool Abstraction [26] metaphor, with a notation designed for our ViTABaL-WS web service composition [53] work, to provide a view to describe event-based tool architecture. This mitigates multiview dependency and consistency issues. TA is a message propagation-centric metaphor describing connections between "toolies" (behavior encapsulations which respond to events to carry out system functions) and "abstract data structures" (ADSs: data encapsulations which respond to events to store/retrieve/modify data) that are instances of "abstract data types" (ADTs: typed operations/messages/events). Connection of

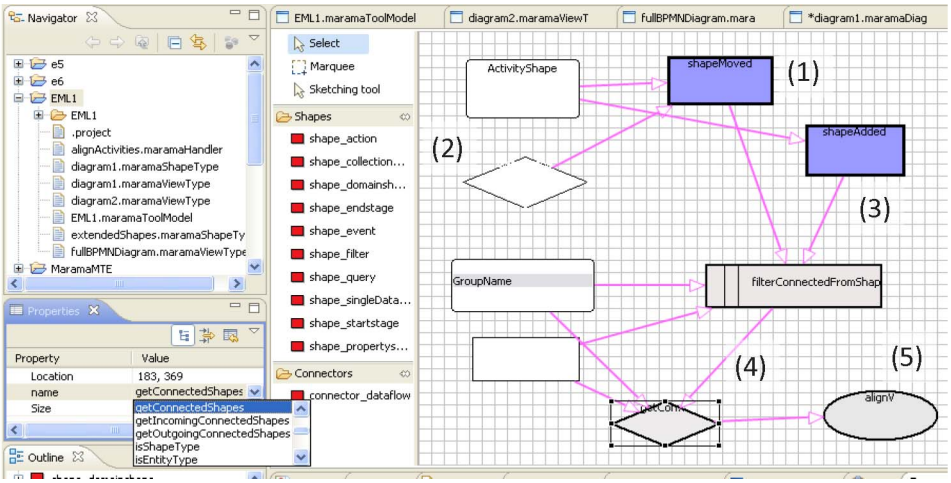



Fig. 11. Specifying an event-driven shape alignment algorithm with the MaramaKaitiaki visual language.

toolies to other toolies and ADSs is via typed ports. TA supports data, control, and event flow relationships in a homogeneous way, allowing a focus on architecture-level abstractions and describing separated concerns including tool specific events, event generators, and receivers, and responding behaviors such as event handlers. Key modeling constructs include event sources/sinks, Marama components, event handlers, toolies (data processing), ADSs (data management), data storage, and error handlers.

Fig. 12 shows specification of user-defined events and propagation of their notifications between various MaramaEML event handling toolies and structural components. This example specifies that when a BPMN diagram is modified and has no constraint violations (1), a BPEL script is generated from the underlying BPMN metamodel entities and associations (2). This BPEL script is then submitted to an LTSA-based model checker (3). This performs a number of consistency checks on the model. Errors detected by the model checking are used to open, annotate, and highlight in the BPMN view (4).

### 5.2.6 Integrated Runtime Visualization Support

Visualization support for a running DSVL tool is also necessary to allow users to track and control system behavior using the same level of abstraction as they are defined in [26], [28]. Synthesized runtime visualization is achieved in Marama via a specialized debugging and inspection tool (a "Visual Debugger"). Our Marama Visual Debugger provides a common user interface that connects the three metaphoric event specification views with an underlying debug model based on the model-view-controller pattern. We use the debugging service instrumentation mechanism [53] to generate low-level tracing events on modeling elements. Marama handles those events by sending the event data to appropriate modeling elements and annotates them with colors and state information. Marama EMF is the common high-level representation that glues different behavioral views together, and supplies dynamic state information to the Visual Debugger. The user has full control of execution, with step-by-step visualization of results (e.g., query results or state changes) at the point of execution of each building block in a particular view. Fig. 13 illustrates the visualization

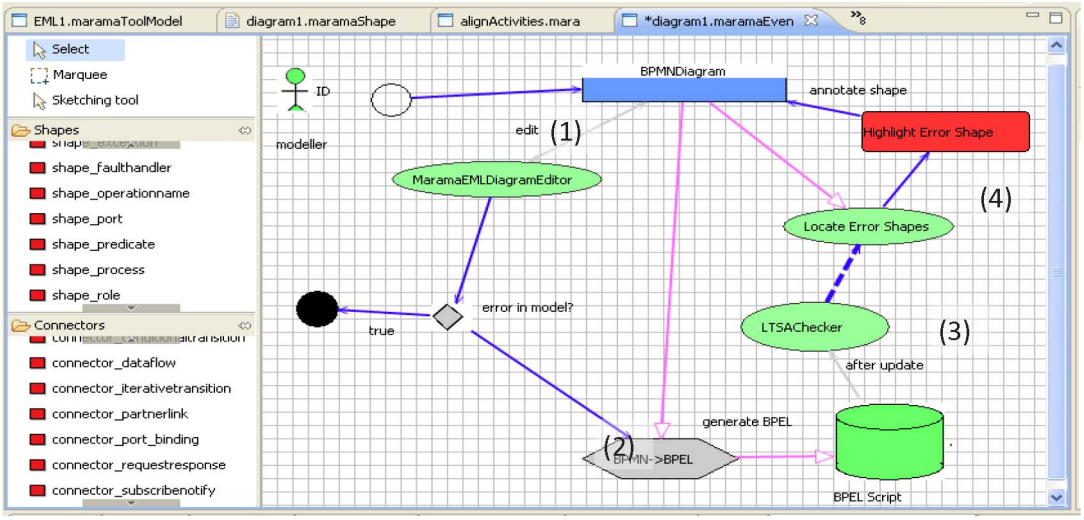



Fig. 12. Event-based tool architecture integration specification.

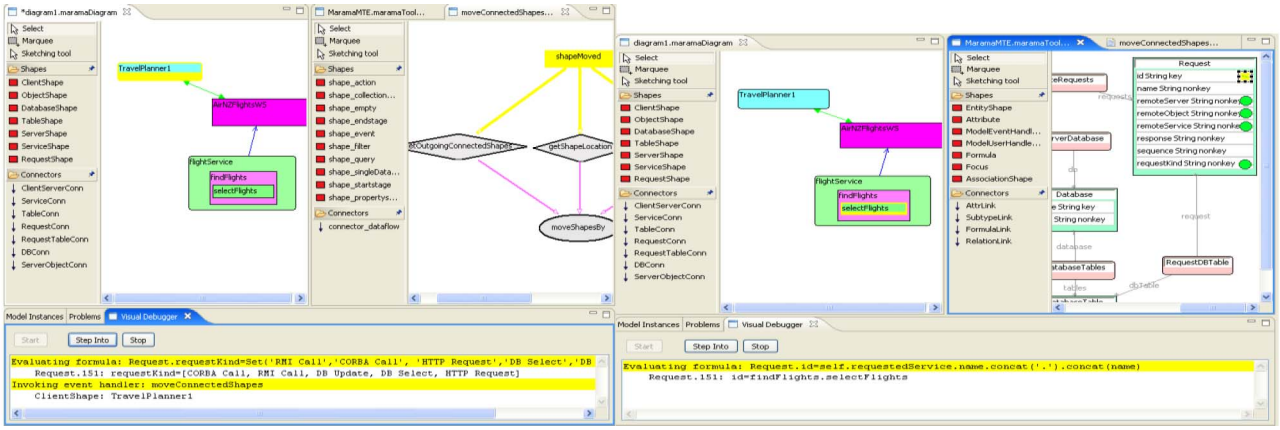



Fig. 13. Visual debugging of a Kaitiaki event handler (left) followed by a MaramaTatau formula (right).

of an event handler (a) followed by a runtime-interpreted formula (b). The Metamodel Designer view and the Event Handler Definer view with the respective formula and event handler specifications are juxtaposed with the runtime diagram modeling view. From the Visual Debugger, the user has control over the execution/interpretation of a behavioral building block. Once the behavior is interpreted, the affected runtime model element is annotated (with a yellow background) to indicate the application of the formula/handler, and meanwhile, the corresponding formula/handler node and their dependency links defined in the corresponding metamodel view are annotated in the same manner to show the behavior specification and its execution status.

## 6 Critics and Transformations (Requirement 3)

Using the Marama capabilities described in the previous section, a tool developer can specify and realize a wide range of DSVL tools. However, our initial Tool Developer evaluations of Marama found two aspects that were problematic: specifying constraints and proactive design advice, and specifying model transformations and code generation. To address these, we developed two further visual specification metatools for Marama: a Design Critic specification tool and a Model Transformation specification tool.

### 6.1 Specifying Design Critics

Research has shown that DVSL tools can greatly benefit from the addition of proactive "design critics" [8]. These critics monitor the state of DSVL tool models and provide proactive feedback to the tool user around model quality. Some critics support "fix up" actions to modify an incorrect or inefficient design structure under user direction. While these can be specified with our declarative and imperative visual event handler specification tools described in Section 5, evaluations of Marama found these to be suboptimal approaches for most tool developers.

To this end, we have developed a visual and form-based critic specification metatool for Marama, MaramaCritic Designer. MaramaCritic provides a high-level, declarative specification approach to add design critics to DSVL tools. It uses a combination of a high-level visual critic model and a more detailed form-based critic specification. Together these provide a critic definition approach that is more accessible, though more restricted in its expressibility, than the other behavioral definition approaches.

The main underlying idea in MaramaCritic is to use information expressed in a metadiagram (i.e., the Marama metamodel diagram) as input for critics to be realized in a diagram (i.e., a Marama diagram in the realized modeling tool specified by the metamodel). It is important to mention that MaramaCritic is only minimally dependent on the notation used in the metadiagram. As we discussed earlier, the Marama metamodel diagram is expressed using an Extended Entity Relationship notation. If a richer notation is used in the future, more information can be extracted from the metamodel diagram and thus can be used for specifying critics and checking user diagrams. Fig. 14(1) shows a simplified metamodel for MaramaEML comprising some of the relevant entities, attributes, and associations. As shown in Fig. 14(1), MaramaEML's main features include service, operation, and process entities. A service entity implies a task within a business process of an organization. An operation entity represents an atomic activity that is included in a service. A process entity has two specializations, process start and process end entities, representing, respectively, the start and end of a process. Associations between the required entities support the modeling of the business process structure. All services, operations, and processes are organized in a tree structure to model a business process system. Fig. 14(2) shows several possible critics for the MaramaEML tool. These specify named design critics to be invoked when various events are generated by Marama model editing operations. Some critics simply provide a "critique" to the user, suggesting problems or issues with the DSVL model state. Others provide "fix up" actions to proactively correct the DSVL model or to enforce constraints on the model expressed in the DSVL. Fig. 14(3) shows a simple example of a MaramaEML structure model for a basic university enrollment service (modified from [51]). Here, the student, university, and StudyLink services are subservices of the university enrollment service. These are represented as oval shapes. Each service may (or may not) include a subservice. The university service includes four embedded services (i.e., enrollment office, finance office, credit check, and department). Each service must

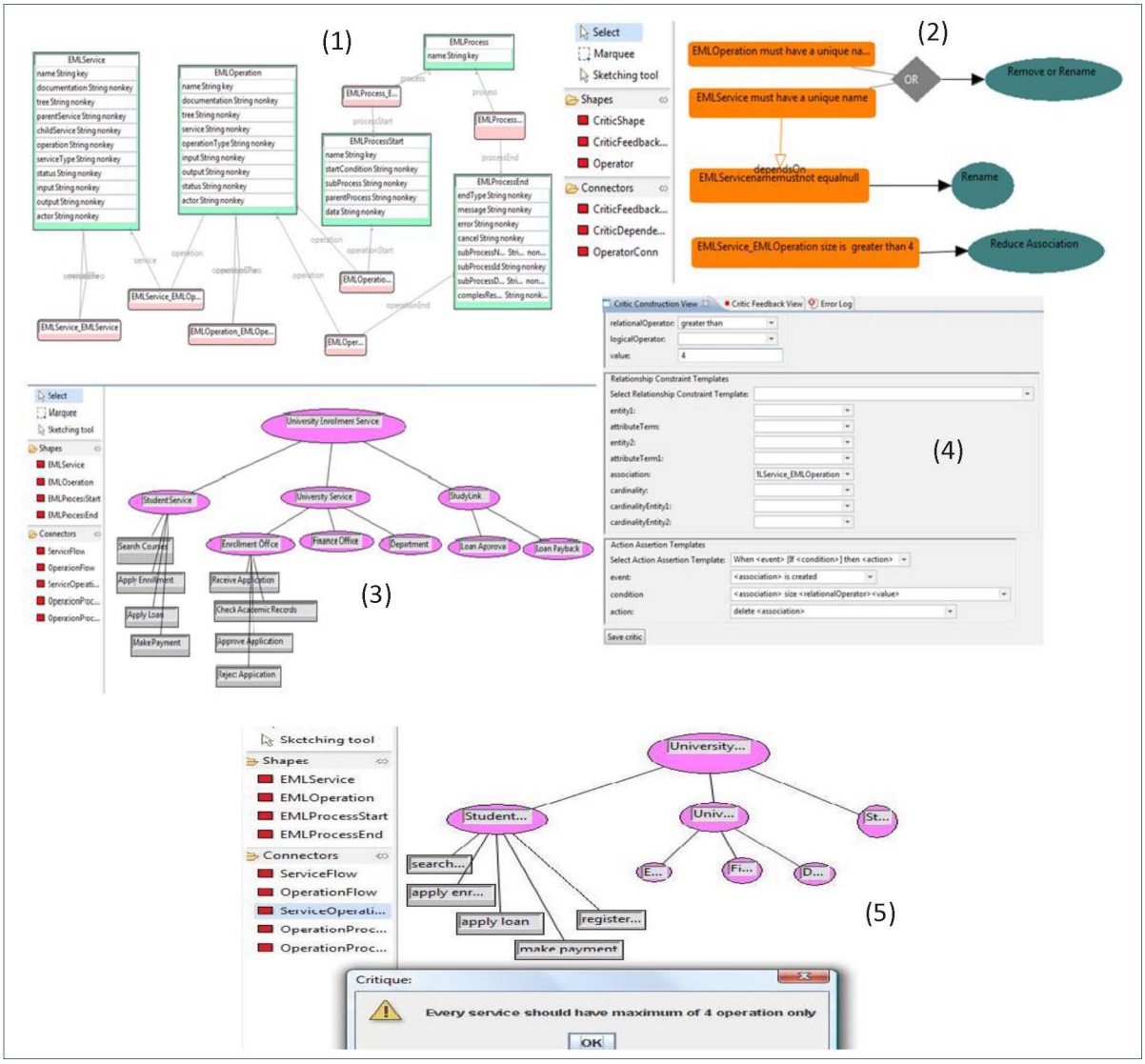



Fig. 14. Metamodel for a simplified MaramaEML (1), followed by a visual and form-based critic specifications (2), an example of MaramaEML structure (3), an action assertion template (4), and critic execution (5).

include at least one operation. The operation entity is represented using a rectangle shape. For instance, the Student Service manages four operations: search courses, apply enrollment, apply loan, and make payment.

The bottommost critic in Fig. 14(2) is an example of an action assertion critic. Suppose the tool developer wants to specify a critic that constrains the service entity (i.e., *EMLService*) to have no more than four operations (i.e., *EMLOperation*). This might be sensible in order to encourage designers to split large hierarchies of services into smaller, more manageable and understandable groups, as our evaluation of MaramaEML found that service entities with large numbers of operations look cumbersome to the tool developers. A critic can be specified for this by defining the relevant properties for event, condition, and action via an action assertion template, as shown in Fig. 14(4)—the form-based interface. Here, the event triggering the critic is the creation of an association link, the condition is that the cardinality is greater than 4, and the action is to delete the new association. MaramaCritic generates, from its visual and form-based specifications, a set of OCL constraints and Java event handlers that augment the generated Marama DSVL tool. When the user runs the tool, these event handlers and constraints are triggered at appropriate times by editing events. A critique (message to the DSVL tool user) is displayed if an event occurs and the model state matches that specified in a critic.

In the arity constraint example, a critique is displayed to warn the user, followed by execution of the action, as shown in Fig. 14(5). Three other critics are specified in Fig. 14(2). The first and second critics at the top specify name uniqueness constraints. A logical operator, OR, is used to link the two critics so that both critics share a common feedback mechanism. The second-to-bottom critic shows a situation where one critic is dependent on another. The dependency of critics can be represented visually by using the CriticDependencyLink connector, which implies a sequence of critic execution between two critics. A critic that depends on another critic will only run when the critic it depends on is not violated. For instance, in Fig. 14(2) it shows the critic: "*EMLService name must have a unique name*" is dependent on a critic: "*EMLService name must not equal*

*null.*" This means the unique name critic is executed only if the service name is not null.

These critics could all have been implemented using Java event handlers to implement similar constraint testing and feedback to the user. However, specifying constraints and feedback using event handlers is time consuming and difficult to maintain as the metamodel evolves over time. Also, as MaramaEML has several integrated modeling notations and a canonical metamodel, it was a complex task to implement internotation constraints. Applying our critic designer to the canonical metamodel is straightforward; implementations of critics that took several hours to specify, test, and evolve using event handlers can be done in a matter of minutes. Understanding the critics is far easier than browsing and understanding the previous individual Java event handlers, which were comprised of hundreds of lines of Java code with Marama API calls. In contrast, the form-based critic specifications are very clear, concise, understandable, reusable, and maintainable. The tradeoff, of course, is that the more accessible notation provides more restricted expressibility.

### 6.2 Model Transformation

DSVL models often need to be transformed. Sometimes transformation is from one model to another, e.g., in MaramaMTE+ we need to transform a Business Process Modeling Notation process flow specification into parts of an architecture specification [25]. We also often need to import information from another format into a tool, such as importing a BPEL specification and transforming it into a BPMN model. To support model-driven development, we often need to generate code, such as Java and C# in MTE+, or scripts, such as JMeter or BPEL. Implementing such transformations using conventional programming languages, or even with higher level transformation DSLs like XSLT, QVT, or ATL, means that users need considerable programming expertise. Originally, Marama tool developers had to write Java code in event handlers to implement code generation and model transforms. We then used a combination of XSLT transforms or ATL transform scripting languages to implement complex code generation and model transformation. Marama tool developer evaluations showed these approaches were very time consuming, error-prone, complex, and difficult to maintain.

To facilitate much more accessible ways to specify model transformation, model import, and code and script generation, we have developed a visual model transformation approach called MaramaTorua [37], now incorporated into Marama as the Transformation Designer. Fig. 15(1) shows an example of MaramaTorua being used to transform a BPMN notation model (left-hand side) into a BPEL4WS executable representation (right-hand side). In the middle is a set of transforms that map source BPMN model elements and relationships into target BPEL4WS elements and relationships. Transforms can be packaged and reused (Fig. 15(2)). Detailed formulas, including element selection, data reformatting, and iteration constructs, are specified using forms (Fig. 15(3)). The visual transform is specified at the type level and is used to generate a transformer implementation. In this example, we generate an XSLT script to implement the specified model transform (Fig. 15(4)). Integrating the generated model transformation into a Marama DSVL tool is achieved by adding a view event handler to a view specification in the relevant View Designer diagram (Fig. 15(5)). The user of the target DSVL tool then selects the transform in a diagram of this view type (Fig. 15(6)) to execute it. MaramaTorua also provides a visual debugger, allowing specified transforms to be stepped through as they are run.

MaramaTorua supports XSLT and Java code generation from its visual model transformation specification models. The visual language and formulas allow quite complex multi-element and attribute model transformations to be generated. It does, however, have limitations when complex transforms need complex algorithmic computation, low-level data parsing, or complex iterative transformations. In this case, MaramaTorua allows Java code to be specified for the transform using an API. Such code is woven into the generated XSLT (as Java function call) or Java code, in much the same way Marama model and view event handler code is generated and added to Marama-specified tools.

## 7 Human-Centric Tool Interaction (Requirement 4)

So far, we have described a set of abstract, visual specification approaches we have developed for Marama metatools to enable high-level specification of DSVL tools. These specifications are used to generate an Eclipse-based implementation of the target tool. However, as identified by Sutcliffe's design metadomain approach, the generated DSVL tools ideally require a range of facilities to support what we term "human-centric" modeling capabilities. These include support for collaborative modeling, sketch-based modeling, and web-based modeling. We have developed a set of plug-in extensions to Marama to support each of these human-centric approaches in Marama-generated DSVL tools. We outline these capabilities in this section to demonstrate the range of human-centric modeling that can be provided for generated DSVL tools. However, while we address some of Suttclife's design metadomain support characteristics, most of these should be viewed as preliminary prototypes requiring further research and experimentation.

To support collaborative use of a Marama DSVL tool, we have developed a distributed support mechanism. This provides asynchronous editing support via visual differencing (MaramaDiffer) and synchronous editing support via event propagation between Marama instances [29], [61]. Fig. 16(1) shows an example of the asynchronous editing support in use. A MaramaMTE+ form chart diagram has been edited by user "john," and user "akhil" is comparing his version of the diagram to john's. Visual annotations are added to the Marama form chart diagram on the right-hand side, showing differences. A change log is shown in the view at the bottom of the screen. Akhil can select to accept all, some, or no changes of john's, and have these applied to his version of the diagram. As some edits are interdependent and some are mutually exclusive, MaramaDiffer provides the user support to merge changes that are consistent. Semantic errors are highlighted in the Eclipse Problems view by appropriate design critics specified for the DSVL tool.

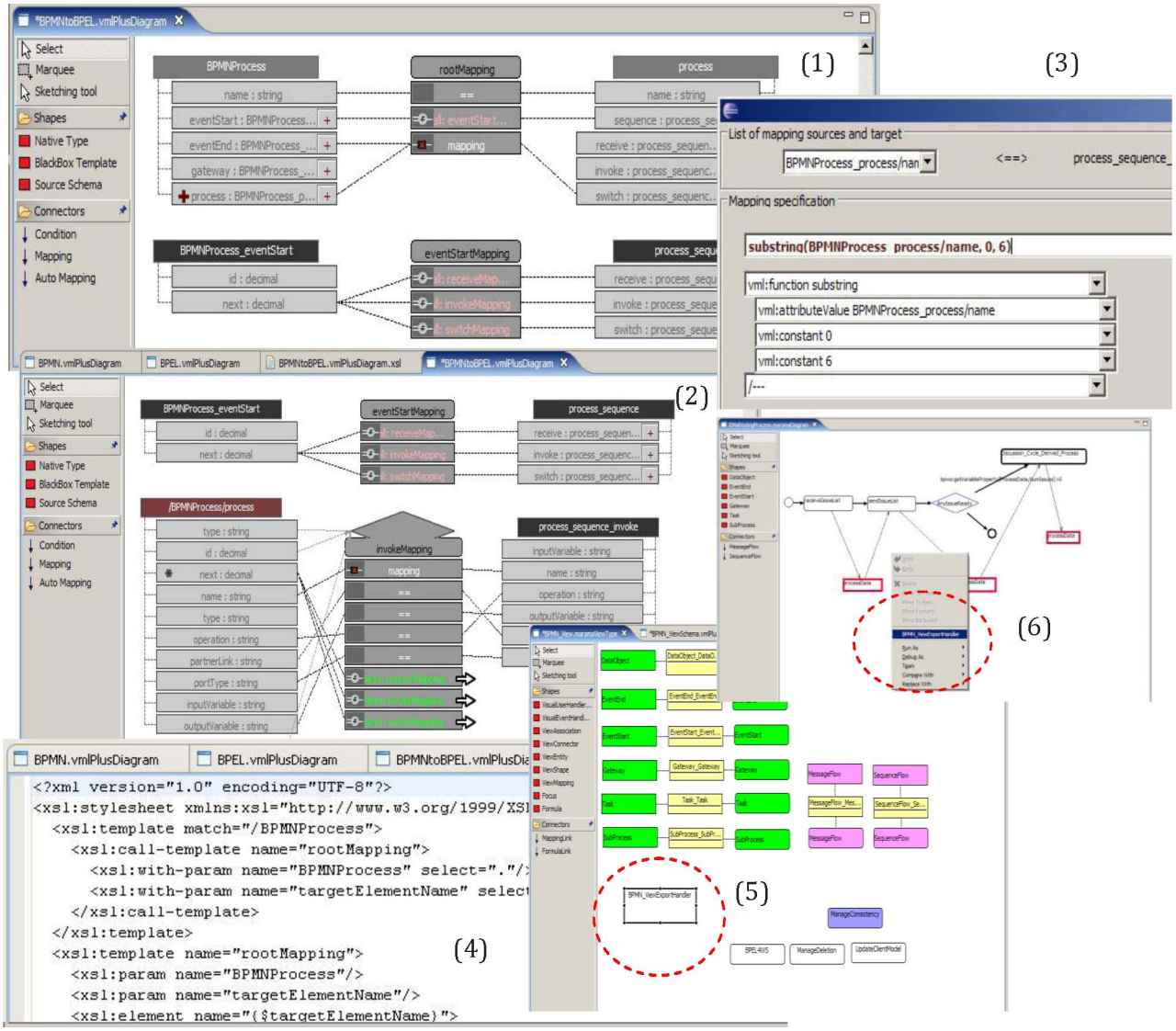



Fig. 15. Model transformation specification.

Sketch-based interaction with design tools is an interesting alternative approach to conventional drag-and-drop diagramming tool interfaces [16], [68]. We wanted to support this approach for Marama design tools as early-phase design has been shown to benefit from this less rigid interaction approach. We developed a new plug-in, MaramaSketch [24], that provides an overlay for Marama diagrams, allowing sketching-based input and manipulation of diagram content along with associated shape and text recognition support. Fig. 16(2) shows an example of a user drawing content (in this example with a Tablet PC stylus) onto a MaramaMTE ArchitectureView diagram. The user simply selects the sketching tool (highlighted in the left-hand side editing palette) and draws with the mouse/stylus on the diagram canvas. In this example, the user has drawn a *ClientShape* (rectangle, "Web UI"), an *ApplicationServerShape* (oval, "Server"), a *DatabaseShape* (cylinder, "DB"), and two connections between shapes. As each set of strokes is completed, MaramaSketch recognizes the shape type and remembers this.

The Marama DSVL tool diagrams shown so far require use of a desktop Eclipse IDE. Rich internet applications do not require desktop application installation but instead provide web browser-based access to information. To make Marama DSVL tools more widely accessible, we have developed a RIA diagramming component, MaramaThin. This is implemented by provision of a set of services within Marama that allow a remote client to query diagram specifications (diagram metadescriptions) and state (diagram model data) via XML messages. The remote clients can also update the state of these diagram models, i.e., modify the diagrams. Fig. 16(3) shows an example of our browser-based diagramming tool MaramaThin in use. User "john" wants to do some ER diagramming for a MaramaMTE+ database. John opens an existing MaramaMTE+ database model diagram. John then begins to edit the diagram. The MaramaThin user interface provides a tool palette, basic editing command toolbar, an ER diagram made up of shapes and connectors that can be manipulated via drag and drop in browser, and pop-up property windows. In this example, John is editing the properties of an Entity "UniName." MaramaThin also provides web-based access to most Marama metatools, including the metamodel editor, shape designer, view type designer, and constraint and event

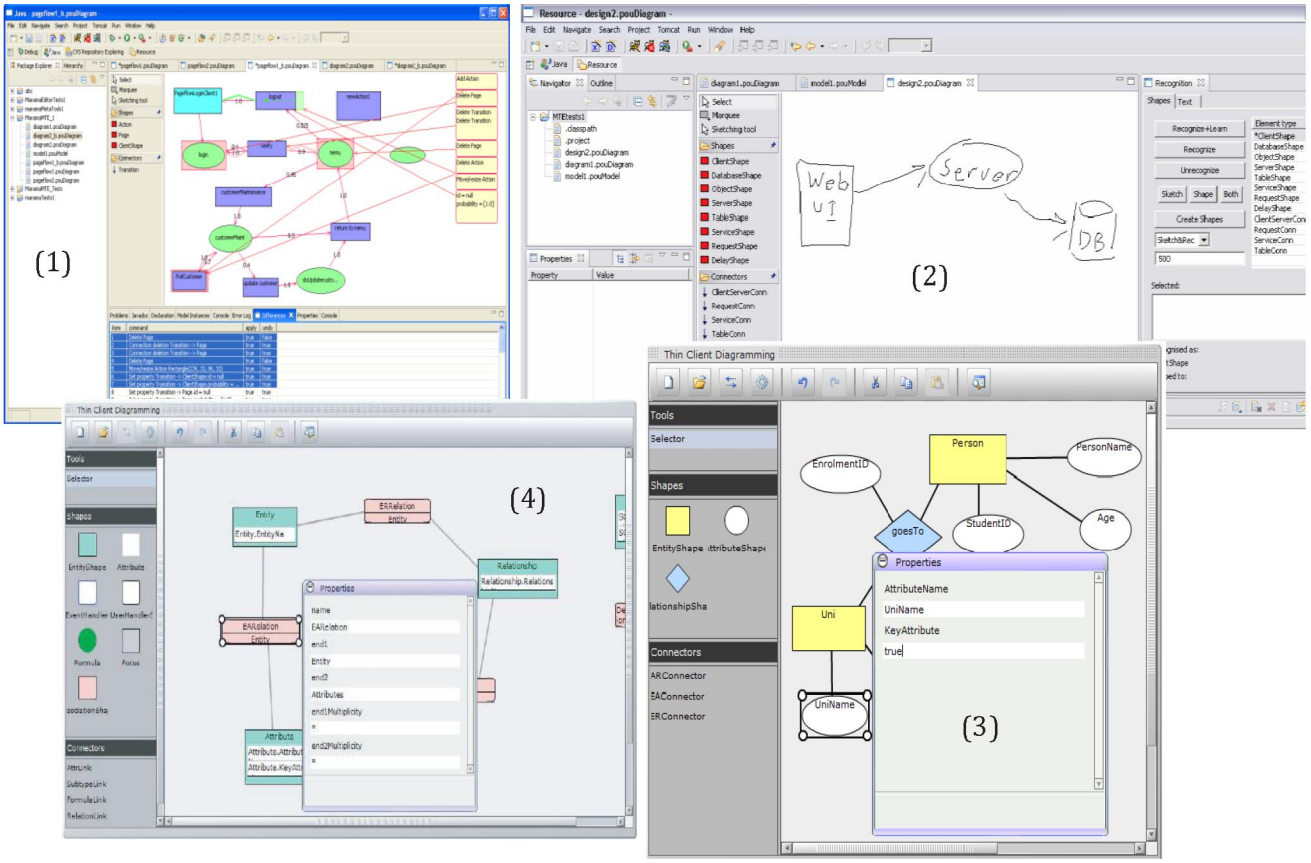



Fig. 16. (1) Collaborative editing, (2) sketch-based input, (3), (4) web-based editing and tool specification.

handler designers. This allows Marama/Thin users to modify their diagram specifications, or even create whole new diagramming tools, using their web browser instead of the Marama Eclipse desktop client. In Fig. 16(4), John is viewing the metamodel for the MaramaMTE+ ER diagrams. He may add new constraints to the tool using the Formula metamodel shape, modify existing constraints, or add a new property, element, or association.

## 8 Architecture and Implementation (Requirement 5)

We have realized Marama as a set of Eclipse IDE-based plug-ins using a range of third-party Eclipse projects and plug-ins. This was a major departure from our earlier Pounamu metatool [79], where we built the whole tool infrastructure. Leveraging the range of Eclipse tool-building projects makes it easier to integrate Marama-generated DSVL tools with other tools, and allows other Eclipse community members to use Marama in their own Eclipse-based tool research.

Fig. 17 shows the architecture of the Marama metatools and Marama-generated DSVL tools. Marama provides a range of visual specification metatools. Each Marama metatool itself provides an editable domain-specific visual language used to specify aspects of a target DSVL tool. Tool developers initially specify a description, comprising evolvable instances of the meta-DSVL models, of their desired DSVL tool using the various visual specification tools in Marama. This DSVL tool specification is opened by a tool user and used to construct an instance of the specified tool. The tool user can then create multiple DSVL models based on this metadescription. Tool developers, and even tool users, can use the Marama metatools to modify a tool specification, often when the tool is "live" (i.e., in use). There are limits to this, e.g., if metamodel descriptions are significantly modified, then an old tool model instance may need to be transformed to the new format before being edited.

Tool specifications are stored as XML documents in a tool specification repository as shown in Fig. 17(1). DSVL tool users locate a desired existing DSVL tool specification to open or request one to be created via the standard Eclipse resource browser (2). When a tool is opened or created in Marama, the corresponding XML tool specifications are read and loaded into XML DOM objects (3). These are parsed and provide an in-memory representation of the Marama tool configuration, which is used to configure an Eclipse Modeling Framework-based in-memory model of both the DSVL tool model and view data, i.e., the properties of all entities, associations, shapes, and connectors. It is also used to produce the editing controls of Eclipse Graphical Editing Framework-based diagram editors, i.e., the allowable shapes and connectors, their rendering, and properties editing (4). When a diagram is opened, Marama configures a GEF editor and renders the diagram (5).

An OCL interpreter is used to implement the MaramaTatau modeled constraints on tool model and view elements (6). The MaramaTatau metatool designer compiles the user-specified constraint formulas into OCL definitions that are stored with the tool specification XML. When a tool is opened,

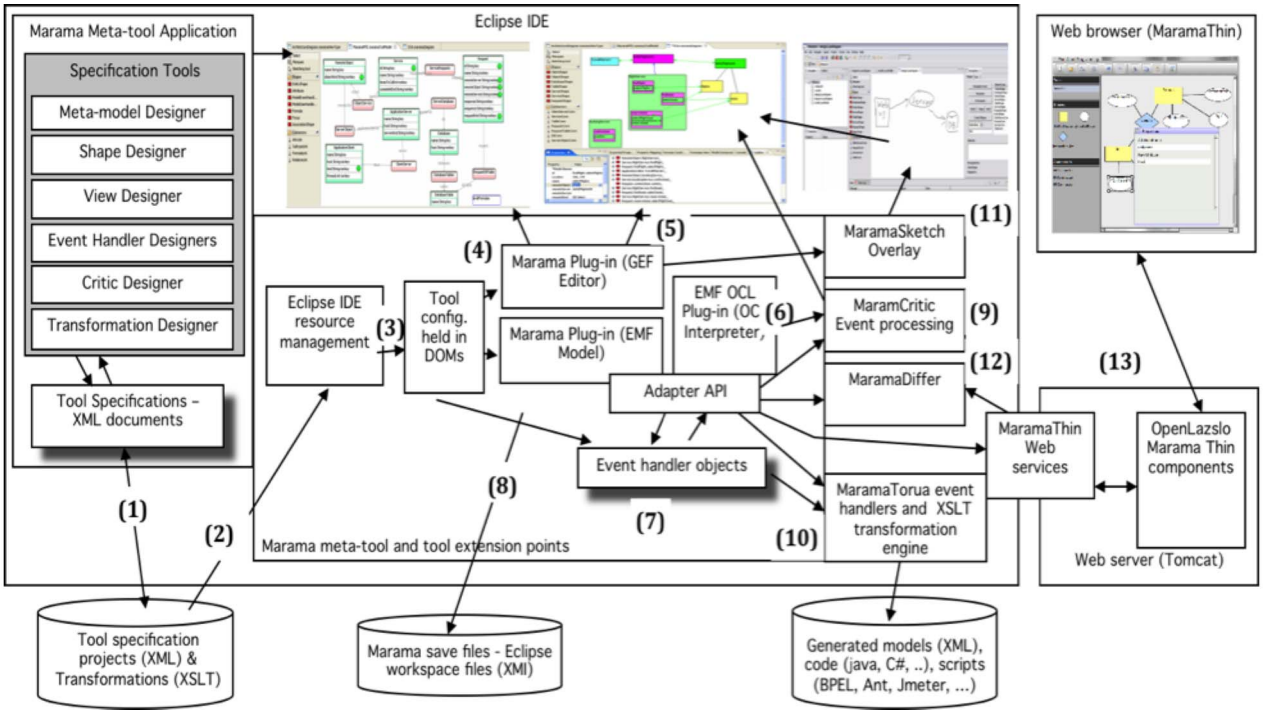



Fig. 17. The basic architecture of Marama.

these OCL definitions are loaded, compiled into the Eclipse OCL API representation used by the Eclipse OCL plug-in and a set of event handlers used to detect changes to model data structures (add/update/delete element or association). These changes are used to trigger recomputation of OCL expressions in much the same way that spreadsheet recomputation occurs. Constraints and user feedback are implemented by Marama API calls triggered by MaramaTatu-inserted OCL expression function calls.

Marama provides a comprehensive API that allows access to all of the Marama data structures and modification of these data structures via Java "event handler" code calls. This Marama API also allows controlled access to the Eclipse GEF drawing APIs and Eclipse plug-in extension mechanism. This allows expert tool developers to extend and augment Marama's core capabilities and support very complex tool integration. Generated event handler and other behavioral and transformational objects are loaded by Marama adapter and Eclipse plug-in APIs when a tool is opened or when the event hander definitions and code are updated in one of the Marama metatools (7). These Java coded event handlers are triggered when Marama model or view data structures are modified or when certain events occur, e.g., button or menu selections or extension point-trigger tool events. Marama uses EMF's XMI save and load support to store and load modeling project data (8), all managed within the Eclipse resource workspace. We have developed a set of extension points for Marama, allowing new metatools to be created and integrated into the Marama metatools suite. We have also created a set of extension points allowing multiple Marama and third-party tools to be integrated within a target Marama DSVL tool framework. Our event-based architectural modeling and integration visual language uses the event handler API and Marama extension points to achieve this tool integration.

Expert tool users can code complex Java event handlers to achieve sophisticated tool behaviors. We also use Java code to implement event handlers that are generated from the imperative event handler specifications from Kaitiaki and MaramaCritic specifications and the declarative MaramaALM specifications. Kaitiaki and our architectural-level event handler specifications generate event handlers to encode their event-condition-query-action models. Typically, tool developers use them to achieve imperative event-based behaviors difficult or OCL-based constraints impossible to code using MaramaTatau. MaramaALM augments the shape and view type definitions and also adds generated event handlers to achieve tree and force-directed layouts for specified view types.

MaramaCritic generates a combination of OCL and Java event handler implementations, augmenting the tool specifications generated by the metamodel and view designer metatools. It also has a set of event handlers (9) used to coordinate processing of events, provide various critiques to tool users (via dialogue boxes and Eclipse problem markers), and carry out semi-automated "fix ups" of problematic model constructs.

The MaramaTorua model transformation metatool generates XSLT transformation scripts, saved with Marama tool specifications, along with Java code event handlers used to invoke these transformations in target DSVL tools (10). These event handlers convert a Marama DSVL tool model to an XML representation via EMF, then use an XSLT engine to transform the EMF structure to the target model (into XML), code (e.g., Java or C#), or script (e.g., Ant, JMeter, BPEL, etc.). An extension mechanism similar to Marama event handlers allows model transformation specifications to include Java code, called from the XSLT scripts, to implement, e.g., low-level data format parsing

and very complex transformations difficult or impossible to specify in MaramaTorua's visual language.

MaramaSketch is a plug-in providing a "sketching overlay" which can be used with any Marama-generated DSVL tool (11). Sketching operations on the overlay are recognized using the HHReco sketch recognition engine [33], which in turn uses a training set of example sketch elements. Sketched items or groups of sketched items are turned into Marama diagram shapes, lines, or text, or groups of these. MaramaDiffer is a plug-in that allows comparing and visual differencing of any Marama DSVL tool diagrams (12). MaramaThin extracts a Marama diagram data structure into XML format using EMF object serialization. It then transforms this into an OpenLazslo-based [7] web-diagramming component, realized using Flash in a web browser to provide a rich Internet client interface (13). MaramaThin uses a set of Java-implemented web services hosted in a webserver (Tomcat) to communicate between the OpenLazslo-implemented diagramming client and the Eclipse-hosted Marama tool instance. MaramaThin uses MaramaDiffer to update the Marama tool diagrams, allowing multiple users to collaborate on diagram editing via a web infrastructure.

MaramaDiffer takes two diagram versions and applies a differencing algorithm to their EMF-based data structures. It determines a set of changes that would transform one diagram into the other and visually presents these as diagram annotations and a list of differences [61]. It provides support for multiple diagram version management, including branching and merging. While this supports diagram merging, it does not fully support model version management, which has limited current support in Marama tools. As tools evolve, their models need to be updated, along with their event handlers. Currently, our Marama prototype provides limited, though not complete, support for this. Basic model version updates can be incorporated, including new entities, relationships, and limited change (renaming) of existing entities and attributes. However, transformation scripts need to be written to transform an old model to a new one for more complex changes, e.g., split/merge entity, move attributes between entities, etc. The MaramaTorua metatool can be used to aid this. Marama event handlers are saved as Java classes and can be versioned using conventional repository check-in/check-out/revision processes. Marama supports simple management of metamodel version and event handler code version. Each Marama metatool model can also be versioned, though limited support for configuration management is currently provided.

## 9 Evaluation

It is not a straightforward task to evaluate a substantial environment/toolset such as Marama as it involves multiple points of views, including those of metatool developer, tool developers of developed tools, usability, and utility, and an enormous number of variables [79]. Most formal usability evaluation approaches are limited to understanding the effect of one or two variables [18], [32]. Controlling for variability is thus an impossible undertaking when assessing the usability of a large environment. We have therefore adopted a variety of less formal, but overlapping approaches to obtain a range of evidence for usability and efficacy. First, we, and our industrial collaborators, have used Marama to construct a range of different modeling tools, some very sophisticated. These provide a proof of concept demonstration that Marama is fit for purpose: It can be used to specify and realize a wide variety of DSVL environments. Second, we have undertaken a variety of formal and informal tool developer evaluations of both the core aspects of the Marama environment and individual component extensions to it. Combined, these evaluations demonstrate both efficacy and tool developer acceptability of our Marama approach.

### 9.1 Experience

We have used Marama to construct a variety of modeling tools in addition to MaramaMTE+, the exemplar used throughout this paper. The Marama metatools themselves (Metamodel Designer, Shape Designer, View Designer, Event Handler Designer, Critic Designer, and Transformation Designer) are all implemented using Marama in a bootstrapped manner and were the first substantial exemplars developed. Four other major model-driven DSVL-based development tools are illustrated in Fig. 18.

MaramaMTE+ [15] is a performance-engineering tool providing two key domain-specific visual languages and a code and deployment script generator. An architectural DSVL is used to model multitier architectures for systems, as illustrated on a simple example in Fig. 18(1). A stochastic form chart DSVL is used to model probabilistic loadings on the defined application. From these models, MaramaMTE+ generates Java or C# code, JMeter or Microsoft ATC load testing scripts, Ant compilation and deployment scripts, database schema and definitions, and various other deployment descriptors for J2EE and .NET applications. It then uses remote agents to run the generated load tests on this model-generated software system, captures the performance results, and visualizes the results in the architecture DSVL diagrams. MaramaMTE+ was part of a PhD project and took two-three weeks to specify and generate the core modeling capabilities using an early version of Marama. The code generators and visualization support took the bulk of the remaining several months work, including experimentation on numerous architectures, target platforms, and load models.

MaramaSUDDEN, Fig. 18(2), provides an environment for integrated supply chain modeling [35]. This allows tool developers, supply chain managers, and SME owners to model and configure supply chains. It includes support for reuse of supply chain models, complex diagram layout control, and integration with external tools to store, validate, and enact supply chain models. MaramaSUDDEN was developed iteratively over a few weeks elapsed as part of a collaborative project. Much of this elapsed time was spent waiting on globally distributed partners to provide feedback on each iteration.

MaramaDPML (Design Pattern Modeling Language) provides a tool to model and instantiate Design Patterns in a UML-based design tool [58]. An example design pattern instantiation diagram is shown in Fig. 18(3), showing the Abstract Factory pattern being instantiated in

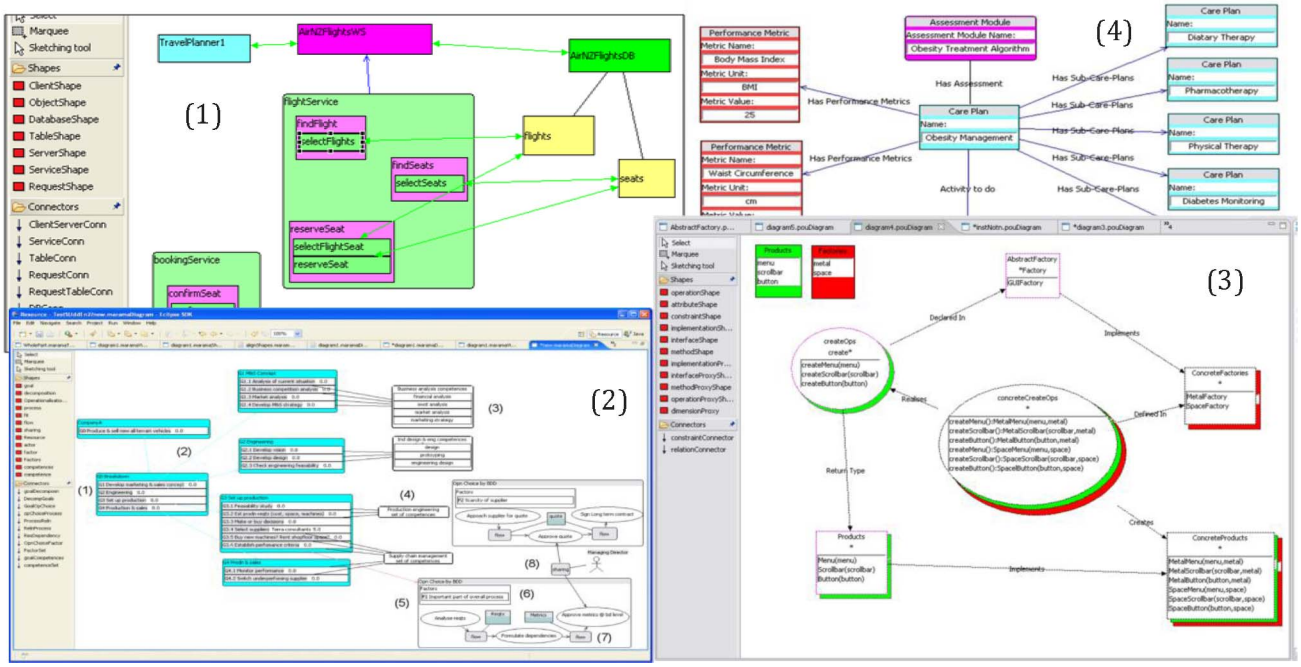



Fig. 18. Examples of Marama generated tools: (1) MaramaMTE+, (2) MaramaSUDDEN, (3) MaramaDPML, and (4) MaramaVCPML.

a UML design for a GUI library project. MaramaDPML provides code generation from the UML models to Java code, multiple, integrated design diagrams with interdiagram consistency management, and validation of design pattern usage in UML designs. Tool developers are software designers. MaramaDPML was developed over a four-five week period based on an earlier prototype using the JViews DSVL tool framework. The main modeling framework took only a few days to specify; the bulk of the implementation time was for back-end code generation, etc.

The final example, Fig. 18(4), is MaramaVCPML (Visual Care Plan Modeling Language) [43]. MaramaVCPML is used to model generic care plans for chronic disease management, which are then instantiated for individual patients. Care plans include modeling of treatments, physical exercise programs, food, and review of key health indicators. Template generic care plan models are reused for multiple patients. MaramaVCPML generates a full mobile device application implementation, allowing patients and their physician to monitor patient progress against their individual care plan. MaramaVCPML supports multiple model integration, code generation, and model template reuse. MaramaVCPML was developed as part of a master's project over a three-four month period.

## 9.2 Formal Evaluations

We have undertaken a significant number of qualitative and quantitative evaluations of DSVL tools developed using Marama [35], [51], [58]. These have been uniformly positive in their overall appraisals of the developed tools. Feedback from tool developers using Marama to produce these tools has also been very positive. We have used Marama over several years in advanced visual language courses, including setting tool development tasks using Marama and surveying the postgraduate students for feedback on core Marama functionality. We have carried out focused qualitative and quantitative studies of individual Marama components to test their usability and effectiveness for specifying DSVL tools and to identify potential problems. We, and a small number of others, have used Marama on several industrial DSVL tool development problems. These evaluation results have been sufficiently positive for us to release Marama as a publicly accessible toolset. We summarize this range of evaluations in the sections below.

### 9.2.1 *Cognitive Dimensions Evaluations of Individual Marama Components*

We have conducted several Cognitive Dimensions [23] evaluations as we developed Marama capabilities to inform the design of the visual meta-DSVL specifications. A CD evaluation provides an understanding of usability tradeoffs and hence where mitigations need to be placed without a need for heavyweight conventional usability evaluations. Typically, as we have developed modeling extensions such as Kaitiaki and MaramaCritic, we have undertaken an individual CD assessment of that extension. These have taken two forms: 1) applying CDs ourselves directly in a similar manner to that proposed by Green and Petre [23] to explore design tradeoffs and 2) using Tool Developer evaluations coupled with a CD-based survey instrument adapted from [10], which reframes CDs into questions understandable by tool developers but which allow a CD analysis of their perceptions to be undertaken.

**Direct CD evaluations.** To illustrate the former approach, consider the Kaitiaki visual handler specification component. In CD terms, we can describe the tradeoffs made as follows (where CD dimension names are in italics): As our target users are inexperienced programmers, we have chosen a low-to-medium *abstraction gradient* based on iconic constructs and data flow between them. The abstractions support query/action composition allowing users to specify Marama data and state changing actions as

discrete, linked building blocks. The abstractions require *hard mental operations* but are mitigated by concrete Tool Developer domain objects. We have experimented with elision techniques to allow concrete icons and abstract event handler elements to be collapsed into a single meaningful icon. The dataflow metaphor used to compose event specification building blocks seems to map well onto users' cognitive perception of the metaphor, providing good *closeness of mapping*. The current approach has reasonable *visibility* and *juxtaposability*. Information for each element of an event handler is readily accessible. The event handler specification can be *juxtaposed* with the modeling view that triggers its execution. However, it still has the usual "box and connector" diagram *viscosity* problems; the user typically has to rearrange the diagram to insert elements.

As another example of this approach, we undertook a similar evaluation for MaramaTatau, the OCL constraint specification designer. In developing MaramaTatau, our focus was on providing a compact and accessible constraint representation for Marama while minimizing *hidden dependency*, *juxtaposability*, and *visibility* issues. 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 a low *abstraction gradient*. Error proneness has been reduced significantly. The preexisting 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 tool developer 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 formulas take effect immediately after reregistration in a tool seveloper 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.

Similar evaluations were undertaken for MaramaSketch and MaramaALM [27], [66], where the emphasis is more on environmental factors than just on the notation. For MaramaSketch, much emphasis was placed on minimizing *premature commitment*, through deferment of recognition of sketch elements, and support of *progressive evaluation* via the ability to recognize on demand. *Viscosity* is much lower than pen and paper sketching equivalents and *closeness of mapping* was central to our motivation, i.e., that sketching is a more natural mechanism for expressing initial designs than standard computer diagramming approaches.

MaramaALM offers a terse notation with simple abstractions (*low abstraction gradient*) and low *viscosity* by encapsulating the low-level implementations into a generalized component that can be easily applied to any Marama-generated tools. It also assists in reducing *viscosity* problems in the generated modeling tools by providing automatic layout in those tools. The tool designers can change the involved shapes and connector in one place and this modification will be reflected throughout the whole mechanism (relatively low *hidden dependencies*). However, the approach comes with the cognitive dimensions tradeoffs of some *hidden dependency* issues and *premature commitment* problems. During the specification process, the use of Shape Designer and View Designer is inseparable due to the structural dependency between both of them. Each depends on the other to generate necessary properties and manage shape-entity mappings in order for MaramaALM to function properly; hence, from one view there is a *hidden dependency* to the other. *Premature commitment* is required as metamodelers need to decide which shapes and connectors are to be included in the layout support during the specification process.

**Cognitive dimensions-based survey-based evaluations.** As an illustration of the second CD evaluation approach, we conducted a user evaluation for our MaramaCritic tool with 12 volunteer researchers and students who had basic background knowledge of the Marama metatools and who were interested in modeling and the development of modeling tools to support their work. The CD-based survey tool provided questions targeted at each of the cognitive dimensions as we were interested in the tradeoffs among those dimensions that respondents observed.

MaramaCritic offers good *visibility* and *viscosity* for the target tool developers. Eleven out of the 12 respondents answered that it is easy to see various parts of the tool and make changes. The only respondent who reported otherwise commented that, due to a lack understanding of metatool concept and as a novice user, it was hard to understand the functionality of various parts of the tool. *Diffuseness* refers to the verbosity of language, i.e., the number of symbols required to express the meaning. Ten respondents reported the notation to be succinct and not long winded. Participants noted that the notation is a straightforward representation of a critic and its feedback, as well as the connectors that link them. MaramaCritic suffers from some *hard mental operations* (degree of demand on cognitive resources): Four respondents claimed that they needed to think carefully about the use of critic templates for specifying a critic. Four respondents disagreed and four were undecided. MaramaCritic is not regarded as being *error prone*, as five respondents claimed that the notation is very straightforward and supported by form-based interfaces that are familiar to most users. The respondents that answered otherwise raised the issue that unfamiliarity with the templates can cause users to make mistakes in specifying critics. MaramaCritic provides good *closeness of*

*mapping.* All respondents noted that the MaramaCritic editor provides a notation closely related to the domain. *Role Expressiveness* for MaramaCritic is obvious as nine respondents answered it is easy to tell what each part is for when reading the notation. Ten respondents said that the dependencies are visible and two respondents are undecided. *Hidden dependencies* are primarily between the visual critic definer view and the form-based template views. Moody argues that this type of hierarchical dependency is of positive benefit in his Principal of Complexity Management [65]. MaramaCritic supports *progressive evaluation* well. Eleven respondents stated it is easy to stop and check work progress. Critics and feedbacks properties can easily be edited and any new changes will take effect during the model execution of the tool. All respondents agreed that there are no *premature commitments* in the Marama Critic Definer view. The user can freely specify a critic using any critic template. However, the user needs to define a critic first before a critic feedback can be specified and linked with the defined critic. The user can add a critic as well as the critic feedback for the Marama tool incrementally as he/she encounters new critics.

The survey results show our respondents have a good degree of satisfaction with our critic design tool integrated with the Marama metatools. The survey results demonstrated that for most respondents our approach appears to be useful in assisting these respondents in the critic specification task. Our approach also appears to nicely complement the other components of the Marama metatools and is integrated with these. However, limitations of the tool are also revealed through the survey results. Thus, some minor improvements are needed to improve the usability of the critic specification editor integrated with Marama metatools. The feedback and limitations that were identified from the survey have led us to refine the current critic specification editor and develop an improved critic specification editor.

#### 9.2.2 Large-Scale Tool Developer Usage of Core Marama Functionality

We used Marama over four years in two final year undergraduate and postgraduate courses. Student tool developers were set a project to build a DSVL tool of their choice and use a range of "core" Marama functionality. We define the "core" Marama functionality as the metamodel designer (excluding constraints), shape designer, view designer, visual OCL editor, and basic event handler specification (nonvisual). These student tool developers documented their work in a short report which included tool description, tool usage example, and reflection on using Marama to develop their DSVL tool. We collated these reports to analyze the range of DSVL tool domains and complexity, usage of Marama features, and feedback on the Marama prototype used. These experiences help us to understand whether student tool developers found the Marama approach and these key Marama metatool features easy and effective for realizing their chosen DSVL tools. We used the tool developers' feedback to improve Marama, and Marama metatool enhancement was undertaken after every iteration.

Participants were 277 graduate-level student tool developers. These were fourth year computer science or software engineering students, i.e., novice short-term research task-oriented users: in 2007—121, 2008—59, 2009—77, and 2010—20. They had used many software tools, though few had used tools for metamodeling similar to Marama. Nothing should be read into the variation in numbers of participants between years: This just reflects cohort availability and time available by the researchers to undertake the DSVL tool development tasks.

These student tool developers were asked to construct a DSVL tool of their choice, but with at least a minimal set of required components so that tools with a realistic level of complexity were developed and participants were required to exploit a core set of the Marama functionality. The required component set included: appropriate numbers of metamodel entity types and associations; appropriate shapes, possibly of differing complexity (of the icon) and connectors; at least two different view types, i.e., showing different types of information within each view type; a few simple event handlers managing things such as diagram layout, editing constraints, model (entity) constraints, mock code generation, data import, etc. Preparatory training sessions were provided on: 1) general DSVL design concepts and principles, 2) a general introduction to metatools and metamodeling concepts, and 3) a specific introduction to Marama architecture and implementation. Student tool developers were also provided with exemplar tools, demonstrations, and tutorials. The tool developers worked individually, in pairs, or in teams of three. The larger the team, the more complex the tool required.

Tool developers were then given three weeks elapsed time (working alongside other commitments) to complete the prototype development, together with an individual survey report answering a set of open-ended questions to qualitatively express both strengths and weaknesses of Marama in constructing their desired DSVL tool. We analyzed the open-ended surveys and categorized the responses into nominal qualitative attributes representing the strengths and weaknesses of Marama from the tool developer's point of view.

A total of 184 DSVL tool instances were developed using Marama in the projects (in 2007—93, 2008—43, 2009—43, and 2010—5). These included, in order of popularity, data modeling (69), component and architecture design (58), process modeling or planning (23), management (22), interface design (7), and requirements modeling (5) tools, summarized in Fig. 19. This indicates that Marama can be effective in application across a large breadth of DSVL application domains. Our set of basic requirements, described above, was met in 69 percent (127 out of 184) of the tools developed. Among those 57 tools that failed to meet all basic requirements, one failed the shape definitions (due to an invalid shape design not reflected in the shape viewers), 33 failed the view type specifications (either omitting a view type or containing unloadable information in the views), and 36 failed the event handler implementation requirements (either not functioning with runtime errors or not implemented at all, with reported reasons including the steep learning curve, difficulty of debugging,

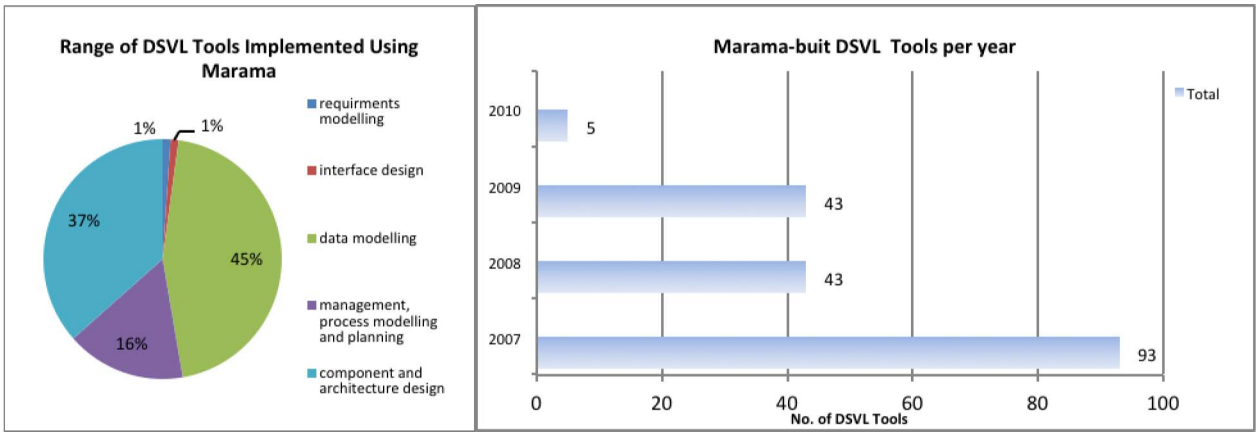



Fig. 19. Broad domains of DSVL tools developed with Marama (left) and tools developed per year (right).

and time constraints). Many struggled with the Marama-Tatau visual OCL constraints in earlier years due to problems with its implementation.

Most tool developers responded in the surveys that Marama was suited to develop their tools effectively (i.e., they were able to achieve the requirements of the assignment) and efficiently (i.e., they were able to achieve their desired DSVL tool results in a "reasonable" time frame) in general. However, there was still room for improvement. We specifically asked students for lists of both strengths and weaknesses, so a comparison of the absolute numbers of strength and weakness responses is not appropriate; however, a categorization of each is instructive.

Fig. 20a shows a broad categorization of strengths identified by tool developers in their reports. Strengths highlighted include: the rapidity of constructing DSVL tools; the simple approach for defining tool data structures with separation of concerns and a seamless integration process; and the consistent, easy to understand metamodel, visual notation and multiple view type abstractions. These reflect the core aims of Marama: the ability to efficiently construct new DSVL-based modeling environments. Lesser numbers of comments related to extensibility and customizability. Comments in reports from tool developers here include ones relating to the powerful event handling mechanism for extending tool behaviors, and the low effort needed and minimum hidden dependency issues arising when customizing and constraining tool developer models and views effectively in the generated tools. The lower numbers here probably reflect that these features are secondary in novice tool developers' minds to those of core metamodel and view definition.

Fig. 20b categorizes the weakness responses. The largest categories concerned the general Marama development environment, and its stability and performance. Participants compared Marama to typical robust open source software and thus expected it to have sound API documentation (access to API documentation is needed for complex event processing) and comprehensive user manuals to help smooth the initial steep learning curve. Neither of these is available to the level of quality desired due to the research prototype nature of the system. Similarly, various modern IDE capabilities were noted to be lacking by different participants. These included automated searching and registration, automatic layout, model validation and refactoring, progress tracking, dynamic debugging, copy/paste, undo/redo, automatic backup and version control, multiple platform portability, and collaboration support. Many of these had been addressed in the research branches of the toolset, but had not found their way into the core branch released to the tool developer participants. Stability and performance issues also irritated tool developers, including the need to: fix a variety of annoying bugs, provide more user-friendly error messages and handle errors more gracefully, rollback unsuccessful operations, reduce memory and resource consumption, and optimize the edit-compile-run and dynamic loading processes. These weaknesses are all typical of the leading edge development of complex software environments and, hence, while irritating, did not give us cause for concern over the viability of our approaches.

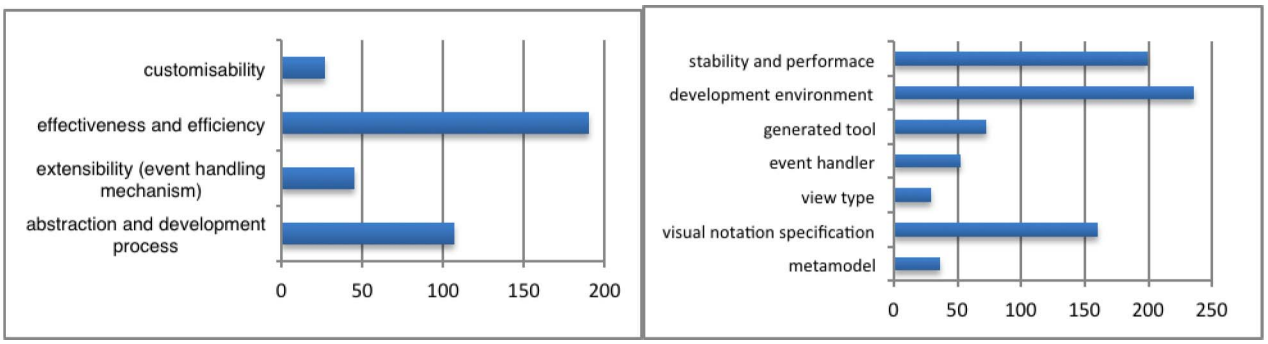



Fig. 20. Strengths of Marama reported from student tool developer reports (left) and weaknesses reported (right).

The most significant "nontrivial" shortcoming of Marama was its limitation for visual notation design. Tool developers found the range of shape/connector types, the flexibility of drawing mechanisms, and the configuration of visual variables (e.g., layout, texture) to be lacking. This category accounted for over half of the remaining identified weaknesses. Respondents also, but to a lesser extent, noted a variety of limitations of Marama-generated tools and suggested including better modeling element identification management; allowing a "smart" connection type that dynamically infers actual types; allowing customizable toolbox items, icons, and tooltips; and allowing deployment of generated tools as standalone applications. Far smaller numbers of comments were made regarding the metamodel, view type, and event handler specification elements. Typical suggestions included providing an n-nary association type, allowing multiple associations between two entities, allowing association subtyping, adding more wizard/dialog or automation support for view mapping; allowing linking of multiple view types passing common values in between (i.e., automapping between metamodel elements), and providing more comprehensive event handler building blocks for composition and reuse. Of note is that most of the nontrivial shortcoming comments were in fact in the form of suggestions for fairly straightforward enhancements, many of which we incorporated into subsequent iterations, rather than fundamental issues with our approach.

We conclude from these experience reports that the Marama approach and core functionality of Marama has demonstrated itself to be a suitable platform for designing baseline DSVL notations, with excellent extension points for behaviors implemented as model/user event handlers. A key aim was to avoid having Marama users need to write complex code and use complex API calls as needed for advanced features in many DSVL toolkits and IDEs. Thus, much of our research focus has been placed on using visual languages and metaphors, such as Kaitiaki, MaramaTatau, MaramaCritic, and MaramaTorua, for event handling, constraint/critique authoring, and model transformation specification. These research features we have evaluated for the most part individually to date.

A few participants each year used an alternative metatool platform to Marama. These included MS DSL Tools (a Visual Studio SDK extension), Eclipse GMF, and MetaEdit+. This was usually because they had previously used Marama on an Honors or Summer Research project. We asked them to report anecdotal evidence of their experiences using these tools compared to Marama. In general, feedback on Marama's core capabilities was positive, with students reporting GMF and DSL tools generally required them to write much more Java or C# code, respectively. They also were required to know many more details of the tool frameworks compared to Marama metatools in order to achieve similar DSVL tool capabilities. MetaEdit+ provided good high-level abstractions, though several of these were form-based or script-based versus visual specifications as in Marama. We plan to have students use and compare other metatool platforms to Marama, such as VisualDiaGen and Tiger, in the future and to more systematically compare and contrast these tools to Marama.

#### 9.2.3 Other Small Scale Evaluations of Individual Marama Components

A further technique we used regularly was to use small groups of users to conduct informal, but usually insightful, evaluations. Participant numbers were typically too small for statistically significant quantitative results but sufficient for qualitative feedback. For example, with the MaramaTorua mapping tool, our Tool Developer evaluation had four experienced data translator implementers carry out a set of mapping tasks (parts of the BPMN->BPEL4WS problem described earlier) [37]. They used MaramaTorua to model the schema, specify a range of mappings, generate an XSLT-based translator, and test it. Overall results were very favorable, with all users able to carry out the task in orders less time than directly implementing Java or XSLT translators. Users expressed overall satisfaction with MaramaTorua's capabilities. Some difficulties found were modeling conditional mappings and string parsing operations and the specification of complex expressions using the MaramaTorua formula editor. Users desired a "design by example" approach for the latter using actual source and target values with the tool inferring conversions. Conducting such simple, fast evaluations regularly informed our development approach.

#### 9.2.4 Industrial Users and Applications

During the development of Marama, we were fortunate to develop a strong relationship with a Model Driven Engineering consultancy company who regularly, over the period of nearly two years, applied Marama to develop industrial strength visual editors that were deployed into large corporate organizations. To support this company, we developed issue reporting and feedback mechanisms, and obtained regular qualitative feedback from the company personnel. While informal in comparison to typical "academic" evaluations, these feedback mechanisms were incredibly useful to us. Tool developers, in this case, were professional programmers and hence differed somewhat from our originally intended target of less technically-able tool developers. As a result, they preferred using the core Marama metatool features of metamodel, shape, view definition, and visual OCL, coupled with handcrafted Java event handlers (i.e., similar to those features predominantly used in our large-scale experience report above) over features more specifically targeted at less technical tool developers, such as Kaitiaki, MaramaCritic, and MaramaTorua. We asked them for feedback on their motivation for choosing to use Marama on industrial projects, their experiences with Marama on these projects, their assessment of the Marama approach and prototype tool platform, and issues that they felt needed to be addressed in future research to improve the Marama approach for industrial adoption.

Motivation for choosing Marama was threefold: its approach to designing DSVLs, its support for modeling with a wide range of generated visual language tools, and its open-source license. Additionally, its use of Eclipse projects and close integration with Eclipse toolsets made it attractive to them. When the company explored options for a next-generation tool for knowledge engineering, they felt very few available tools had these characteristics. Marama was used by the company on two significant projects, each

involving development of a set of visual languages/editors and was used in demonstrators along with generated DSVL tools on a wide variety of occasions. Significant development of Marama was undertaken by the company, in collaboration with our research group, to "industrially harden" the Marama prototype implementation. This allowed for more scalable and robust Marama tools to be developed and deployed, as well as making the Marama metatools implementations themselves more robust.

Overall, the feedback provided on the core Marama features, while anecdotal, was extremely positive, with high satisfaction reported by the tool developers on the productivity afforded using Marama. Also reported was strong satisfaction by tool end-users when Marama generated tools were deployed into their client organizations. Some of the key "shining aspects" of Marama reported by the company included its approach to supporting generation of DSVL editors using predominantly visual specification techniques "without confusing technical overheads" and with good separation of concerns. The ease of turning conceptual models into visual diagramming tools often "wowed" industrial partners of the company and the end users of the tools. The company judged the Marama approach was significantly ahead of other platforms such as MetaEdit+, DS Tools, and GMF.

Some limitations of Marama related to its research prototype nature, the immaturity of its implementation and a range of usability limitations, some due to its reliance on Eclipse projects, and some, as also reported by our student tool developers above, due to its prototypical nature. Many potential users lacked confidence to develop Marama-based tools either on their own or exclusively relying on external parties to assist in these efforts. Much of this lack of confidence related to a lack of reusable patterns/diagrams/parts, something we have been addressing in recent work [49]. Some lack of custom graphic symbols, out-of-the-box layout managers, and reliance on Eclipse deployment approaches also drew negative comment from potential users. The greater platform and usability maturity of other DSVL tools, such as MetaEdit+, was significantly attractive to some tool developers.

However, for this high-end tool developer, more fundamental limitations with the Marama approach, and realization of the approach in our current prototype, were also identified. These included the lack of a shared, robust, and highly scalable repository for digital artifacts. This was a major requirement for many model-centric organizations. The desire for an entirely web-based user interface for both specification and generated modeling tools was also important. Additionally, a clean implementation of a denotational semantics that completely decouples modeling from naming, and a desire for a more mathematically based metamodel including model and category theory underpinning the modeling framework was desired by our partner, rather than using less formal EER metamodels as currently used by Marama. These requirements are well beyond those of our target tool developer community for which a simple, readily understandable metamodel approach is preferable. Marama currently also lacks textual DSL tool integration and has no direct support for textual DSL design and generation. Many organizations adopting model-driven engineering use textual DSLs and wanted this support to be as accessible and integrated in Marama as its current support for DSVLs.

We have also used Marama ourselves on several industrial projects, predominantly developing proof of concept tools to assist in our own consulting work or to assist our industry partners in their R&D projects. Significant Marama tool projects whose results fed into industrial R&D efforts included developing an XForms designer, two business process modeling tools, a supply chain modeler, a health care plan modeler, and a requirements engineering support tool. All of these tools, developed with Marama, were used experimentally by industrial partners and their target end users. As with our partner experiences above, we found Marama to be very effective for designing new domain-specific visual languages and exploratory authoring of these by end users. The generated DSVL tools could be rapidly developed and experimented with. However, limitations with its model repository, limited support for large model visualization, minor but annoying usability issues, and lack of textual DSL integration all contributed to these tools not being widely deployed.

## 10 Discussion

Combined together, our experiences gained from complex DSVL application development, the core environment evaluations, the individual component evaluations, and our industrial users indicate our Marama approach to DSVL tool engineering is effective. Our small-scale evaluations separate concerns, allowing focus on individual components, and have provided evidence for their efficacy and usability. Our more substantial whole of environment evaluation provides evidence that the components work effectively when combined together. In both cases, good evidence for the efficacy and usability of Marama has been provided. Deficiencies noted, particularly from the large-scale evaluation, are predominantly related to expected software maturity issues, such as a lack of API documentation and software stability, and issues specific to nontarget user groups, such as the need for a more mathematically robust metamodel, rather than fundamental issues with the approach. Experience from applying Marama to realize substantial DSVL environments, both academic and industrial, allows extrapolation from the more formal, and hence restricted, evaluations to more realistic usage. Both the developer experience in specifying these environments and the tool developer evaluations of their usability and efficacy provide strong support for Marama's effectiveness as a metatool.

While our aims in developing Marama were to afford metatool capability to nontechnical tool developers to allow them to develop their own modeling environments, we are not quite at the point of generating sufficiently robust evidence to demonstrate that we achieved this. Our evaluations have focused on more technically proficient tool developers to date. Conducting similar studies on nontechnically proficient audiences remains future work.

We have, however, met each of the key requirements for a DSVL metatool we established in Section 2. Looking in more detail at Sutcliffe's Design metadomain model [73], we see that Marama provides strong support in some areas

and partial support in others, suggesting future work opportunities as follows:

- Modeling and prototyping support; visualization support:

Marama's multiple metaphor meta-DSVL models provide effective separation of concerns by specifying different DSVL aspects at different abstraction levels. At a tool level, there is seamless integration of these multiple abstractions; however, hard mental operations and hidden dependencies are introduced as a tradeoff for the resulting specification flexibility. Users need to decide which visual language to use at a particular modeling stage. Therefore, there is a need for a description and guidelines for these metaphors from which users can make better choices about their specification approaches. An obvious direction to proceed is to use MaramaCritic to specify a set of metacritics for the Marama metatools embodying such high-level guidelines and constraints. We aim to operationalize Moody's Physics of Notations [65] design principles in these critic-based guidelines and to provide some basic visual language assessment support for evolving DSVLs within Marama itself.

In addition, the expressibility provided by some of the Marama metatool components is currently limited, making it difficult to design and realize some types of diagram. Specifically, modeling tools are limited to box and connector, with some limited containment support. Such restrictions are common in current metatools. We are currently looking to extend this to allow for notations such as Euler graphs [36] which combine box and connector and overlapping region requirements.

To provide better and more integrated support for information presentation and visualization, we aim to merge modeling with visualization, to empower visualization with the capability of extracting/generalizing models for reuse. From this, instead of creating new models from scratch each time, we will allow users to explore existing models and capture reusable components by various visualization functionalities such as querying, filtering, and abstracting. This extends our Kaitiaki work with more complex reusable query and visualization support [54].

- Critics and knowledge retrieval/reuse:

MaramaCritic provides good support for specification and realization of tool level critics. These, however, need to be manually specified by tool developers, whereas Sutcliffe envisaged some form of automated reasoning over prior solutions. Similarly, Marama currently has very limited support for reuse of designs and part designs. A promising direction we are exploring is metapattern support, which provides facilities for representing and instantiating domain-specific and domain-independent metafragments [40], [49], including a set of generalized/generic tasks (e.g., formating trees, juxtaposing multiple view display). This will provide one step toward a more intelligent approach to design reuse.

- Annotation, collaboration, and creativity support:

While Marama has some useful annotation, collaboration, and creativity support via its collaborative editing and web and sketch-based diagramming support, there is room for enhancement. We plan to employ program-by-demonstration techniques to provide ways of recording, simulating, and validating designs. Such techniques should allow users to play prerecorded macros to learn the visual languages and their modeling procedures, and to specify their own domain systems following demonstrated examples or patterns. In addition to the current procedure of generating DSVL environments from metalevel structural and behavioral specifications, we wish to also allow users to demonstrate the intent of their DSVL tools and automatically reverse generate the specifications (both structural and behavioral) from that, with further refinement allowed via round-trip engineering.

## 11 Conclusions

Models are used in a huge range of domains. This provides an obvious driving force for good tools to author, visualize, manage, and evolve models. We have described Marama, a meta-DSVL tool for multiview DSVL tool generation. The core of Marama comprises a set of visual meta-DSVL models for specifying both the structural and behavioral aspects of DSVL environments. Extensions include design critics, model transformation, collaborative editing, thin-client, and sketch-based editing interfaces. We have developed a number of complex DSVL tool applications in various domains using Marama. These include performance engineering, enterprise/business process modeling, design patterns, health care planning, data modeling, software engineering, and so on. We have carried out a variety of evaluations of Marama itself and Marama-developed DSVL tools. Overall, while there are a number of areas for further enhancement of the metatoolset, it provides an effective set of DSVL-based metatools for designing and realizing a wide variety of DSVL tools. Key enhancements proposed include using design critics to provide "metacritics" and constraints for DSVL language design, support for knowledge reuse via DSVL patterns and refactoring support, and further extension and enhancement of annotation, collaboration, and human-centric modeling interfaces. The Marama toolset has now been released in an open source form (https://wiki.auckland.ac.nz/display/csidst/Welcome) and is being taken up by a number of research groups and industrial partners for software tool prototyping. It is in the process of commercializing and "industry hardening" with SofismoAG.

## Acknowledgments

The authors gratefully acknowledge the support of their colleagues, Assoc. Prof. Robert Amor, Dr. Beryl Plimmer, Dr. Gerald Weber, Dr. Rainbow Cai, and many project students and industry partners. They also acknowledge funding provided by The New Zealand Ministry of Science and Innovation's Software Process and Product Improvement project, the Malaysian Government, the University of Auckland, and the Tertiary Education Commision's funded BuildIT project.

## References

[1] "Eclipse Graphical Editing Framework," www.eclipse.org/gef, 2012.

[2] "Eclipse Graphical Modelling Framework," http://www.eclipse.org/gmf/, 2012.

[3] "IBM ILOG JViewsDiagrammer," http://www-01.ibm.com/software/integration/visualization/jviews/diagrammer/, 2012.

[4] "Merlin Generator," http://www.sourceforge.net/projects/merlingenerator/, 2012.

[5] "Microsoft Domain Specific Language Tools," http://code.msdn.microsoft.com/vsvmsdk, 2012.

[6] "NetBeans Visual Library," http://platform.netbeans.org/graph/, 2012.

[7] "OpenLaszlo," http://www.openlaszlo.org/, 2012.

[8] N.M. Ali et al., "End-User Oriented Critic Specification for Domain-Specific Visual Language Tools," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 297-300, 2010.

[9] R. Bentley et al., "Supporting Collaborative Information Sharing with the World-Wide Web: The BSCW Shared Workspace System," *Proc. Fourth Int'l World Wide Web Conf.,* 1995.

[10] A.F. Blackwell and T.R.G. Green, "A Cognitive Dimensions Questionnaire Optimised for Users," *Proc. 12th Ann. Meeting Psychology of Programming Interest Group,* 2000.

[11] S. Bottcher and S. Grope, "Automated Data Mapping for Cross Enterprise Data Integration," *Proc. Int'l Conf. Enterprise Information Systems,* 2003.

[12] F. Brieler and M. Minas, "A Model-Based Recognition Engine for Sketched Diagrams," *J. Visual Languages and Computing,* vol. 21, no. 2, pp. 81-97, 2010.

[13] J. Buchner, T. Fehnl, and T. Kuntsmann, "HotDoc a Flexible Framework for Spatial Composition," *Proc. IEEE Symp. Visual Languages,* pp. 92-99, 1997.

[14] M. Burnett et al., "Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm," *J. Functional Programming,* vol. 11, no. 2, pp. 155-206, 2001.

[15] Y. Cai, J. Grundy, and J. Hosking, "Synthesizing Client Load Models for Performance Engineering via Web Crawling," *Proc. 22nd IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 353-362, 2007.

[16] Q. Chen, J. Grundy, and J. Hosking, "An e-Whiteboard Application to Support Early Design-Stage Sketching of UML Diagrams," *Proc. IEEE Symp. Human Centric Computing Languages and Environments,* pp. 219-226, 2003.

[17] P.T. Cox et al., "Experiences with Visual Programming in a Specific Domain—Visual Language Challenge '96," *Proc. IEEE Symp. Visual Languages,* 1997.

[18] A. Dillon, "Usability Evaluation," *Encyclopedia of Human Factors and Ergonomics,* W. Karwowski, ed., Taylor and Francis, 2001.

[19] D. Draheim and G. Weber, "Modeling Submit/Response Style Systems with Form Charts and Dialogue Constraints," *Proc. Workshop Human Computer Interface for Semantic Web and Web Applications,* pp. 267-278, 2003.

[20] J. Ebert, R. Süttenbach, and I. Uhe, "Meta-CASE in Practice: A CASE for KOGGE," *Proc. Int'l Conf. Advanced Information Systems Eng.,* A. Olivé and J. Pastor, ed., pp. 203-216, 1997.

[21] K. Ehrig et al., "Generation of Visual Editors as Eclipse Plug-Ins," *Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 134-143, 2005.

[22] D. Gordon et al., "A Technology for Lightweight Web-Based Visual Applications," *Proc. IEEE Conf. Human-Centric Computing,* pp. 28-31, 2003.

[23] T.R.G. Green and M. Petre, "Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework," *J. Visual Languages and Computing,* vol. 7, pp. 131-174, 1996.

[24] J. Grundy and J. Hosking, "Supporting Generic Sketching-Based Input of Diagrams in a Domain-Specific Visual Language Meta-Tool," *Proc. 29th Int'l Conf. Software Eng.,* pp. 282-291, 2007.

[25] J. Grundy et al., "Performance Engineering of Service Compositions," *Proc. Int'l Workshop Service-Oriented Software Eng.,* pp. 26-32, 2006.

[26] J.C. Grundy and J.G. Hosking, "ViTABaL: A Visual Language Supporting Design by Tool Abstraction," *Proc. IEEE Symp. Visual Languages,* pp. 53-60, 1995.

[27] J.C. Grundy and J.G. Hosking, "Serendipity: Integrated Environment Support for Process Modelling, Enactment and Work Coordination," *Automated Software Eng.,* special issue on process technology, vol. 5, no. 1, pp. 27-60, 1998.

[28] J.C. Grundy, J.G. Hosking, and W.B. Mugridge, "Visualising Event-Based Software Systems: Issues and Experiences," *Proc. Software Visualisation Workshop,* 1997.

[29] J.C. Grundy et al., "Generating Domain-Specific Visual Language Editors from High-level Tool Specifications," *Proc. 21st IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 25-36, 2006.

[30] J.C. Grundy, W.B. Mugridge, and J.G. Hosking, "Constructing Component-Based Software Engineering Environments: Issues and Experiences," *J. Information and Software Technology,* vol. 42, no. 2, pp. 117-128, 2000.

[31] E. Guerra, J.D. Lara, and P. Diaz, "Visual Specification of Measurements and Redesigns for Domain Specific Visual Languages," *J. Visual Languages and Computing,* vol. 19, no. 3, pp. 399-425, 2008.

[32] H.R. Hartson, T.S. Andre, and R.C. Williges, "Criteria for Evaluating Usability Evaluation Methods," *Int'l J. Human-Computer Interaction,* vol. 15, no. 1, pp. 145-181, 2003.

[33] H. Heloise and A.R. Newton, "Sketched Symbol Recognition Using Zernike Moments," *Proc. 17th Int'l Conf. Pattern Recognition,* 2004.

[34] R.C. Holt, A. Winter, and A. Schurr, "GXL: Toward a Standard Exchange Format," *Proc. Seventh Working Conf. Reverse Eng.,* 2000.

[35] J. Hosking, N. Mehandjiev, and J. Grundy, "A Domain Specific Visual Language for Design and Coordination of Supply Networks," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing,* pp. 109-112, 2008.

[36] J. Howse, P. Rodgers, and G. Stapleton, "Drawing Euler Diagrams for Information Visualization," *Proc. Sixth Int'l Conf. Diagrammatic Representation and Inference,* p. 4, 2010.

[37] J. Huh et al., "Integrated Data Mapping for a Software Meta-Tool," *Proc. Australian Software Eng. Conf.,* pp. 111-120, 2009.

[38] G. Johnson, *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control,* second ed. McGraw-Hill School Education Group, 1997.

[39] G.E. Kaiser et al., "WWW-Based Collaboration Environments with Distributed Tool Services," *World Wide Web,* vol. 1, no. 1, pp. 3-25, 1998.

[40] M. Kamalrudin, J. Hosking, and J. Grundy, "Improving Requirements Quality Using Essential Use Case Interaction Patterns," *Proc. 33rd Int'l Conf. Software Eng.,* 2011.

[41] S. Kelly, K. Lyytinen, and M. Rossi, "Meta Edit+: A Fully Configurable Multi-User and Multi-Tool CASE Environment," *Proc. Int'l Conf. Advance Information Systems Eng.,* 1996.

[42] R. Khaled et al., "A Lightweight Web-Based Case Tool for Sequence Diagrams," *Proc. SIGCHI-NZ Symp. Computer-Human Interaction,* 2002.

[43] A. Khambati, J.C. Grundy, J.G. Hosking, and J. Warren, "Model-Driven Development of Mobile Personal Health Care Applications," *Proc. 23rd IEEE/ACM Int'l Conf. Automated Software Eng.,* Sept. 2008.

[44] C.H. Kim, J.G. Hosking, and J.C. Grundy, "A Suite of Visual Languages for Statistical Survey Specification," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing,* pp.19-26, 2005.

[45] G.E. Krasner and S.T. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *J. Object-Orientated Programming,* vol. 1, no. 3, pp. 26-49, 1988.

[46] J. Lara and H. Vangheluwe, "AToM3: A Tool for Multi-Formalism and Meta-Modelling," *Proc. Int'l Conf. Fundamental Approaches to Software Eng.,* R.-D. Kutsche and H. Weber, eds., pp. 174-188, 2002.

[47] A. Ledeczi et al., "Composing Domain-Specific Design Environments," *Computer,* vol. 34, no. 11, pp. 44-51, Nov. 2001.

[48] T. Levendovszky et al., "A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS," *Electronic Notes in Theoretical Computer Science,* vol. 127, no. 1, pp. 65-75, 2005.

[49] K. Li et al., "Augmenting DSVL Meta-Tools with Pattern Specification, Instantiation and Reuse," *Proc. Workshop Visual Formalisms for Patterns,* vol. 31, 2010.

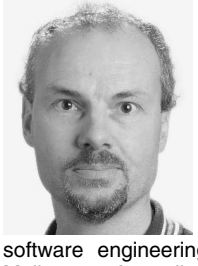
[50] K.N.L. Li, J.G. Hosking, and J.C. Grundy, "A Generalised Event Handling Framework," *Proc. Workshop IEEE/ACM Automated Software Eng. Conf.,* 2009.

[51] L. Li, J.C. Grundy, and J.G. Hosking, "EML: A Tree Overlay-Based Visual Language for Business Process Modelling," *Proc. Int'l Conf. Enterprise Information Systems,* 2007.

[52] Y. Lin, J. Gray, and F. Jouault, "DSMDiff: A Differentiation Tool for Domain-Specific Models," *European J. Information Systems,* vol. 16, no. 4, pp. 349-361, 2007.

[53] N. Liu, J.C. Grundy, and J.G. Hosking, "A Visual Language and Environment for Composing Web Services," *Proc. ACM/IEEE Int'l Conf. Automated Software Eng.,* 2005.

[54] N. Liu, J.C. Grundy, and J.G. Hosking, "A Visual Language and Environment for Specifying User Interface Event Handling in Design Tools," *Proc. Eighth Australasian User Interface Conf.*, 2007.
[55] N. Liu, J.G. Hosking, and J.C. Grundy, "MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, 2007.
[56] M. Lyu and J. Schoenwaelder, "Web-CASRE: A Web-Based Tool for Software Reliability Measurement," *Proc. Int'l Symp. Software Reliability Eng.*, 1998.
[57] D. Mackay, J. Noble, and R. Biddle, "A Lightweight Web-Based Case Tool for UML Class Diagrams," *Proc. Fourth Australasian User Interface Conf. User Interfaces*, vol. 18, pp. 95-98, 2003.
[58] D. Maplesden, J.G. Hosking, and J.C. Grundy, "A Visual Language for Design Pattern Modelling and Instantiation," *Design Patterns Formalization Techniques*, T. Taibi, ed., Idea Group, Inc., 2007.
[59] F. Maurer et al., "Merging Project Planning and Web Enabled Dynamic Workflow Technologies," *IEEE Internet Computing*, vol. 4, no. 3, pp. 65-74, May/June 2000.
[60] D.W. McIntyre, "Design and Implementation with Vampire," *Visual Object-Oriented Programming*, pp. 129-159, Manning Publications Co., 1995.
[61] A. Mehra, J.C. Grundy, and J.G. Hosking, "A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.*, 2005.
[62] M. Minas, "Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation," *Science of Computer Programming*, vol. 44, no. 2, pp. 157-180, 2002.
[63] M. Minas, "Visual Specification of Visual Editors with VisualDiaGen," *Applications of Graph Transformations with Industrial Relevance*, J.L. Pfaltz, M. Nagl, and B. Böhlen, eds., pp. 473-478, Springer Berlin/Heidelberg, 2004.
[64] M. Minas, "Generating Meta-Model-Based Freehand Editors," *Proc. Third Int'l Workshop Graph Based Tools*, 2006.
[65] D.L. Moody, "The 'Physics' of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering," *IEEE Trans. Software Eng.*, vol. 35, no. 6, pp. 756-779, Nov./Dec. 2009.
[66] Y.S. Pei, J.G. Hosking, and J.C. Grundy, "Automatic Diagram Layout Support for the Marama Meta-Toolset," *Proc. IEEE Symp. Visual Languages and Human-Centric Computing*, Sept. 2011.
[67] C. Phillips et al., "The Design of the Client User Interface for a Meta Object-Oriented CASE Tool," *Proc. Technology of Object-Oriented Languages* , 1998.
[68] B. Plimmer and M. Apperley, "INTERACTING with Sketched Interface Designs: An Evaluation Study," *Proc. CHI '04 Extended Abstracts on Human Factors in Computing Systems*, pp. 1337-1340, 2004.
[69] L. Qiu and C.K. Riesbeck, "An Incremental Model for Developing Educational Critiquing Systems: Experiences with the Java Critiquer," *J. Interactive Learning Research*, vol. 19, pp. 119-145, 2008.
[70] J. Rekers and A. Schuerr, "Defining and Parsing Visual Languages with Layered Graph Grammars," *J. Visual Languages and Computing*, vol. 8, no. 1, pp. 27-55, 1997.
[71] J.E. Robbins and D.F. Redmiles, "Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML," *J. Information and Software Technology*, vol. 42, no. 2, pp. 79-89, 2000.
[72] C. Schuckmann et al., "Designing Object-Oriented Synchronous Groupware with COAST," *Proc. ACM Conf. Computer Supported Cooperative Work*, pp. 30-38, 1996.
[73] A. Sutcliffe, *The Domain Theory: Patterns for Knowledge and Software Reuse.* L. Erlbaum Assoc., 2002.
[74] J.M. Vlissides and M. Linton, "Unidraw: A Framework for Building Domain Specific Graphical Editors," *Proc. Ann. ACM Siggraph Symp. User Interface Software and Technology*, pp. 158-167, 1989.
[75] P. Wohed, W.M.P. van der Aalst, M. Dumas, A.H.M. terHofstede, and N. Russell, "On the Suitability of BPMN for Business Process Modelling," *Proc. Fourth Int'l Conf. Business Process Management*, pp. 161-176, 2006.
[76] J. Zhang, Y. Lin, and J. Gray, "Generic and Domain-Specific Model Refactoring Using a Model Transformation Engine," *Research and Practice in Software Eng.*, vol. 2, pp. 199-218, 2005.
[77] K. Zhang, D.-Q. Zhang, and J. Cao, "Design, Construction, and Application of a Generic Visual Language Generation Environment," *IEEE Trans. Software Eng.*, vol. 27, no. 4, pp. 289-307, Apr. 2001.
[78] C. Zhao, J. Kong, and K. Zhang, "Program Behavior Discovery and Verification: A Graph Grammar Approach," *IEEE Trans. Software Eng.*, vol. 36, no. 3, pp. 431-448, May/June 2010.
[79] N. Zhu et al., "Pounamu: A Meta-Tool for Exploratory Domain-Specific Visual Language Tool Development," *J. Systems and Software*, vol. 80, no. 8, pp. 1390-1407, 2007.



**John C. Grundy** received the BSc(Hons), MSc, and PhD degrees, all in computer science and all from the University of Auckland, New Zealand. Previously, he was a lecturer and senior lecturer at the University of Waikato, New Zealand, and a professor of software engineering and head of electrical and computer engineering at the University of Auckland, New Zealand. He is currently a professor of software engineering and head, computer science and software engineering, at the Swinburne University of Technology, Melbourne, Australia. He is an associate editor of the *IEEE Transactions on Software Engineering*, *Automated Software Engineering* journal, and *IEEE Software*. His current interests include domain-specific visual languages, model-driven engineering, large-scale systems engineering, and software engineering education. He is a member of the IEEE and the IEEE Computer Society.

**John Hosking** received the PhD degree in physics from the University of Auckland before joining the academic staff there. He is a dean of engineering and computer science at the Australian National University. He progressed through to the rank of professor and had six years as the head of the department before taking up his role at ANU. He has research interests in software tools, software engineering, and visual languages and environments. He is a fellow of the Royal Society of New Zealand and a member of the IEEE and the IEEE Computer Society.

**Karen Na Li** received the PhD degree, followed by postdoctoral research at the University of Auckland from 2004-2011, specializing in visual software development notations and techniques. She is now working in industry, practicing visual programming (in particular the use of IBM technology stack) for client software development, through which she is gaining more insight into visual software techniques.

**Norhayati Mohd Ali** received the PhD degree in computer science from the University of Auckland, New Zealand, in 2011. She is a senior lecturer in the Information System Department, Faculty of Computer Science and Information Technology, Universiti Putra Malaysia. Her research interests include critiquing tools in software engineering, visualization in software engineering, software engineering education, and human-computer interaction.

**Jun Huh** has worked as a research assistant for Professor John Hosking and Professor John Grundy since 2005, specializing in model-driven visual development. He has been heavily involved in developing prototypes using and for the Marama metatools. He is currently working as a researcher and developer in Visual Wiki project, which involves visualization of web-based information and knowledge.

**Richard Lei Li** is the Enterprise Information Services Manager at Beef+Lamb New Zealand. This role, directly reporting the Chief Operating Officer, is to design the enterprise level information management strategy and lead the development of technical solutions. His expertise predominantly is in business process modeling and technology integration and transformation, but also covers human-computer interaction, visual languages and information management strategy.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.