Increasing the Flexibility of Modelling Tools via Constraint-Based Specification

Philip Gray and Ray Welland

Department of Computing Science, University of Glasgow, UK
{pdg,ray}@dcs.gla.ac.uk

Abstract

Most commercial modelling tools provide support for customising surface features (i.e., visual and, to some extent, interactive behaviour) of a model. Although useful and simple to use, such customisation is typically very limited; for example, one cannot change the basic representation of model components. Meta-CASE tools offer the potential for much greater customisation, but at a high cost, viz., the tool must be respecified and rebuilt. We propose an approach, constraint-based specification, which combines the flexibility of current tools with the power of CASE tool builders.

1 Introduction

A meta-CASE tool is a tool that provides facilities for specifying and generating CASE tools [1]. A common use of such tools is to generate design diagram editors and our examples will be drawn from this domain. The use of meta-CASE, or tool builder, technology provides flexibility in the generation of graphical design tools, extensibility to provide tools for new, hybrid or integrated notations and methods, the efficient use of the graphical infrastructure underpinning many different tools. Our objective is to provide finer control over a generated tool's interaction, providing greater flexibility of interaction, giving individual users more control of some aspects of a tool's behaviour, particularly with respect to enforcement of design rules.

Varying the specification used as the basic input to the meta-CASE tool provides flexibility in the generation of design tools. For example, to provide a local variation of a widely used design notation, such as UML, we can modify the specification of the standard notation and generate a new variant of the design tool. However, it would not be economic (nor manageable!) to generate individual variants of tools tailored for individual users.

Run-time customisation of tools is possible within the limits set by the tool designer; any more substantial changes will require the tool to be rebuilt. Run-time customisation is a form of configuration, but it can also be viewed as micro-evolution of the tool specification. The limitation of most current tools is that the degree of control over run-time customisation is ad hoc. A user can change just those features that the tool provider anticipated should be tailorable. Normally, these customisation facilities are purely cosmetic, allowing the use of different colours or changing the line routing in a diagram. There is no general model of what is changeable or why. Our work combines the use of meta-CASE technology with run-time constraints (described below) to create a two level configuration model for tools, the coarse level being provided by the meta-CASE tool and the fine level of control via the use of run-time constraints.

This paper reports on recent work to combine independent but related research on software modelling tools and constraint-based user interface development tools. We examine the ways in which

constraints can be exploited in meta-CASE tool technology and develop a three-level characterisation of configuration constraints for modelling tools. The potential of this approach is demonstrated via several examples of tool customisability and the limitation of some current tools. We propose a constraint-based meta-CASE system that provides greater flexibility and customisability of software modelling tools. An outline of our proposed architecture is presented, followed by a discussion of related work and concluding observations.

2 Specification and Constraints

2.1 Specification via Constraints

The notion of constraint is based on the notion of value dependency. That is, some data element e is constrained by condition c if the value of e depends on the value of c. Constraint specification languages have proved useful for a variety of applications in which value dependencies are volatile and subject to change. A number of successful user interface development environments, for example, have been implemented using constraints to specify the interactive behaviour of graphical elements [4, 6].

Software modelling tools, such as those generated by meta-CASE tools, are similar in relevant ways to user interface development environments:

- both generate end-user systems featuring interactive graphics, and
- as we have argued above, support for customisation is a desirable feature.

The power of the constraint approach lies in the fact that constraints can be specified via constraint languages and interpreted at run-time via constraint-satisfaction engines. Changes to the constraints (e.g., to reflect customisation) can be carried out without rebuilding the system; the new constraints will be resolved by the actions of the constraint-manager.

We find it useful to distinguish several varieties of constraint from the point of view of tool design and construction:

2.1.1 Design-time Constraints

Design-time constraints are restrictions on system behaviour identified or asserted by a designer; typically they express functional and non-functional requirements. They may be expressed in many different forms or, indeed, may merely be informal requirements existing only in the head of the designer.

2.1.2 Executable Constraints

These are constraints that affect the behaviour of an actual system. They may be hard-coded in the system implementation, in which case they are embedded constraints, or they may be run-time constraints that are resolved by a constraint-manager. Ideally, the executable constraints will capture all and only the set of design-time constraints.

Implementation of a tool can be viewed as transforming design-time constraints into executable constraints (see figure 1).

In the simplest case, where meta-CASE technology is not used, tool specification will be expressed in a programming language, the toolbuilder will be a compiler and all of the executable constraints will be implicit in the executable image (i.e., they are ”embedded” in the code).

2.2 Embedded Constraints vs. Run-Time Constraints

A major part of the specification of the design notation that is used as input to a meta-CASE tool is a set of design-time constraints on the way in which the generated tool can be used. For example, the type or number of connections permitted between node types may be restricted; names may be required to be unique over certain classes of diagram entities; the graph must be connected, no disjoint partitions of the graph are allowed. These constraints may be embedded directly in the code of the run-time system or explicitly separated out as conditions to be handled by a constraint manager. Together these (executable) constraints are used to enforce the rules of the diagramming notation.

In a conventional meta-CASE tool that does not generate a constraint-resolution subsystem in the output tool, any executable constraints must, of course, be embedded constraints. Furthermore, some meta-CASE specification languages do not

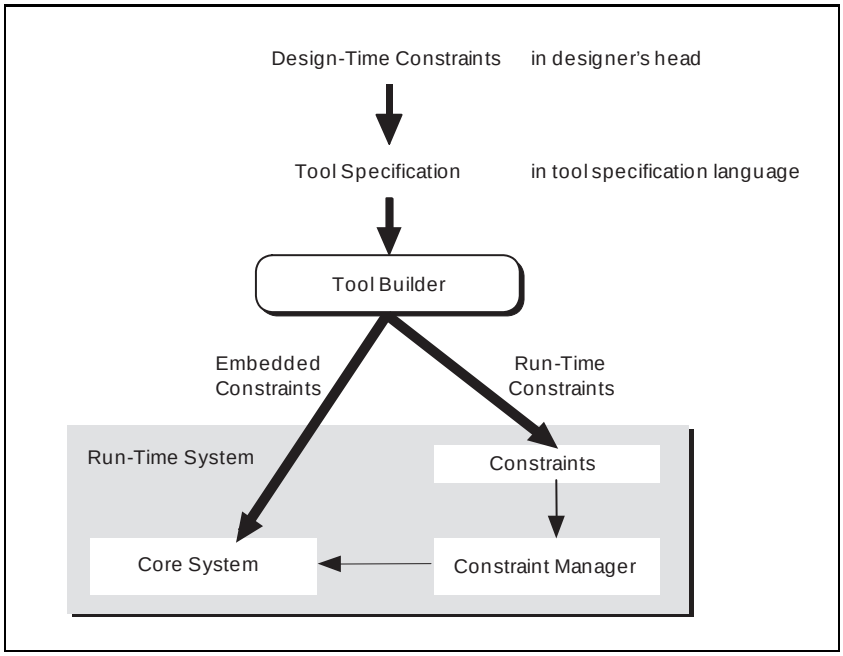

Figure 1: Relationships Among Constraint Types

offer the option to describe constraints explicitly as conditions to be met by a diagram. Design-time constraints can only be expressed in such languages by embedding them in the constructs that define the syntax of the design notation. However, in the case of a system that permits run-time constraint checking, the question arises: which constraints should be embedded and which should be run-time? We now turn to that question.

2.3 Hard vs. Soft Constraints

the way in which the user can interact with the generated tool. We can classify executable constraints as hard constraints, which are enforced as a diagram is drawn, and soft constraints, which may be temporarily violated by intermediate states of the diagram. For each constraint there is a choice about when it should be enforced and in a standard meta-CASE tool this decision is hard-wired into the generated tool. Soft constraints will eventually have to be satisfied before a legal design diagram is created; their enforcement is simply deferred until either the user requests checking or checking is invoked automatically. Given a suitable constraint-satisfaction engine, run-time constraints offer the potential to modify the constraints from hard to soft and vice versa. That is, the time of constraint enforcement can itself be treated as customisable. This additional customisation can be quite useful: hard constraints can be relaxed to speed up or simplify complex software modelling and soft constraints can be hardened to check the current status of the software model.

2.4 Why Restrict Constraints to Customisation?

We are unlikely to generate an editor in which all the design-time constraints are implemented as run-time constraints. This type of 'free for all' editor would be grossly inefficient and would make it very difficult to give sensible feedback to the user, as there would probably be cascades of dependent errors. Therefore, some of the constraints will be embedded and the tool designer will have to decide where the dividing line between embedded and run-time constraints should be drawn. This could be seen as balancing compiling constraints and interpreting them. Typically, hard constraints will be compiled into actions enforced by the generated tool (i.e., embedded constraints), while soft

constraints will be interpreted at run-time. Hard constraints will normally be related to the operations that the tool allows. For example, attempting to make an illegal connection between two nodes will be prohibited, i.e. implemented as an embedded hard constraint, if there is no operation that allows the insertion of a new node to transform the current state into a legal diagram.

Our approach is to separate run-time constraints from the rest of the specification of the design notation so that a constraint engine can process them at execution time. In this way, the end-user can decide at what point in a diagram development process a particular soft constraint should be enforced. At one extreme, the user can decide that all run-time soft constraints should be enforced immediately (i.e., treated as hard constraints), giving a syntax-directed style of editing. At the other extreme, all run-time constraints can be relaxed (i.e., treated as if they were soft) giving the effect of a compilation system in which all checking is post hoc. In practice, we expect most users to adopt an intermediate position, identifying their own subset of constraints to be enforced immediately, leaving the remainder of the constraints to be post checked.

3 Configuration Constraints Applied to Modelling Tools

3.1 Types of Tool Configuration

Three basic types of specification or configuration can be identified in graph-based modelling tools:

- changes to the actual graph structure, which we call the relationship level;
- changes to the information associated with nodes and arcs, called the representational level;
- changes to the visual appearance of the graph and its associated information, called the presentation level.

We refer to these three types of customisation as levels since they form an interpretational hierarchy. That is, relationship level customisation concerns the structure of the uninterpreted graph, representational level customisation refers to the interpretation of the graph, and presentational level customisation affects the mapping of the interpreted graph onto visual properties of a display.

3.1.1 Relationship Level

At this level, properties of the graph are specified. Characterisation at this level affects the graph structure, but has no effect on the meaning of the graph components or their visual presentation. Thus, a STN might be customised so that cycles are not permitted; that is, the graph must be acyclic.

This type of condition on the modelling technique should be distinguished from using the graph structure for investigating properties of an actual model specification. That is, one might want to assert that there exists a path between two specified nodes (e.g., to prove that an interactive sequence can be performed). Such a condition is not on the modelling technique but on a particular use of the technique. While such conditions can be handled using the approach described below, they are beyond the scope of the present paper.

It is perhaps misleading to say that the relationship level only affects a completely uninterpreted graph. Often graph properties are specified by reference to the type of the nodes or arcs. Thus, consider a tool for constructing petri nets. This might be customised for use in specifying user interfaces by declaring a special node type, called a device place, which represents the source of device-generated input events. One might require that such places have no input transitions. The condition, no input transitions into device places, applies to the graph structure, but refers to the typing (hence interpretation) of the nodes. Our three-level categorisation of configuration is based on what is affected by the configuration, not on the type of information in the constraint condition.

3.1.2 Representation Level

Configuration at this level conditions the meaning of graph components while preserving the underlying graph-based relationships among components. One may think of this as augmenting the graph structure with additional information. That is, one may customise nodes and arcs by placing conditions on the information associated with them. For example, one might require that nodes representing transforms in a DFD must have a unique name. The customisation does not change the graph prop-

erties of the DFD but only affects the information associated with nodes.

3.1.3 Presentation Level

Configuration at this level affects the visual appearance of the model, including the graphical rendering of model components and the manner of their layout.

This may seem the least important of the three forms of configuration, but the visual presentation can be critical for the understandability of the information in the model. A poor layout (e.g., many arc crossovers in a graph) or graphical components breaking well-known conventions (e.g., using diamonds to stand for entities in an ER diagram) can increase the difficulty of "reading" the model and cause errors of interpretation.

3.2 Capturing a Modelling Technique

Customisation at all three levels may be combined with a base-level tool specification to define a modelling technique. Consider the configuration of a DFD tool. At one level of abstraction, a data flow might be defined as a labelled relationship between a source transform and a destination transform. We can describe customisations at all three levels of a tool related to this abstraction:

- relationship level: a data flow is represented by a directed arc between transform nodes.
- representation level: the data flow must have a label (not necessarily unique).
- presentation level: the data flow is presented as a line with a textual label centred on the centre of the line and an arrow pointing to the destination transform.

A lower level of abstraction, in which the data flow label is a named and typed data element, could modify the representation and presentation levels of the tool.

This multi-level description of a modelling technique can be defined by a set of appropriate run-time constraints. We can think of these like "style" specifications that can be applied to a basic DFD modelling tool [8]. Rebuilding of the tool is unnecessary to change the modelling technique, only the set of constraints defining the style need be changed.

3.3 Capturing and Supporting Design Guidelines

Software design is a demanding intellectual activity, often helped by design knowledge (e.g., principles, guidelines and local expertise). Typically, software modelling tools do not support the design process, apart from supplying an interactive specification medium and reducing the effort of recording the results of the process.

Constraints can encapsulate design knowledge expressed in guidelines like:

- don't put more than 5 classes in a diagram
- put inputs on the left, outputs on the right
- where appropriate, prefix names with the parent entity's abbreviated label (or other such local naming convention)

Such guidelines provide additional assistance in producing software models that are readable and likely to lead to satisfactory designs. However, it is possible for legal diagrams (in the sense of expressing a model with correct semantics) to violate these guidelines.

Soft run-time constraints can be used to capture such design guidelines in the modelling tool itself. Violations of the guidelines might trigger an "error" which can lead to a subsequent design review. We believe that one of the strengths of our approach is the ability to handle guidelines as well as definitional features of a modelling technique.

4 An Example

We present an example of our approach based on a popular modelling technique, viz., class diagrams in UML [14]. We show how constraints can be used to customise all three levels of a modelling tool. First, by changing the constraints on inheritance relationships, the graph structure of the tool is modified. Second, by constraining the class representation, we customise the attributes of a class. Finally, we alter the visual appearance of the class representation by constraining the visual display of class members.

4.1 UML Class Diagrams

A UML class diagram is a description of the types of objects in an object-oriented system. Included in

such a diagram are visual representations of

- the name, attributes and operations of a class, and
- various static relationships among the classes, including inheritance and aggregation.

Class diagrams can be used to generate viewpoints (i.e., descriptions at different levels of abstraction), including:

- class relationships only, with classes represented by boxed names
- architectural information only, with classes represented by their external interface (public state and operations)
- implementation information, where class representation also include private state and methods.

4.2 Relationship Constraints

The UML modelling notation places few restrictions on the class descriptions that can be produced. Thus, there is no facility in the notation for distinguishing single from multiple inheritance nor is there requirement that class descriptions must, or must not, include certain information about class members (e.g., that attributes must be typed). It is reasonable to suppose that restrictions such as these would be useful for certain software modelling tasks. We shall now examine how tool configuration might support configuration to capture such restrictions.

Suppose I, a modeller, want to restrict my specification to single inheritance between classes. To reflect that condition, inheritance relationships must be such that each class has at most one parent. There are three options to reflect this condition in a modelling tool:

- if the tool is not reconfigurable and does not support such a condition, then the tool user must maintain the condition mentally by ensuring that no class node is given more than one incoming inheritance arc. This is clearly undesirable since the user may forget or overlook a violation of the condition and, in any case, an additional cognitive load is placed on the user
- if the tool allows for modification, the operation for creating inheritance relationships might be rewritten and the tool rebuilt. While this is clearly better than the first option, the overhead involved in the rebuild is rather great. Also, the only representation of the condition is the modified code itself, making it difficult to validate the correctness of the change
- if the tool allows for specification of such conditions in a high-level constraint language, the single inheritance condition can be expressed in terms of the features of the augmented graph, enabling validation. Such a language offers the opportunity, in principle, to generate a new tool satisfying the constraint or to add the constraint at run-time, thus reducing the cost of rebuilding by automating code generation.

In UML class diagrams produced using the Rational Rose modelling tool [11], one may add several different types of relationship: inheritance, association, aggregation, etc. As Rational Rose is presently constituted, a ”single inheritance” constraint cannot be specified at the level of the tool; it must be maintained by a careful designer. Hence, Rational Rose only offers the first option.

4.3 Representation Constraints

A designer might well want to modify class representations or add new ones. For example, I might want a class representation that captures only name and a set of attributes (abstractions over state and operations) as in the JavaBeans approach [10]. To do this in Rational Rose, a convention would have to be utilised by which special public state variables stand for the attributes and operations are disallowed in the specification. As with the relationship constraints described above, maintaining the constraint is the responsibility of the tool user.

In addition to the reasons already mentioned, this approach is clearly inadequate, since it does not capture the notion of different levels of abstraction. If a tool supported modelling constraints, the ”attributes only” version of a tool might be changed to an ”attributes and attribute-related operations” version just by changing the representational (and perhaps presentational) constraints. This provides

a method for the systematic transformation or replacement of the one abstraction by another.

4.4 Presentational Constraints

Tools like Rational Rose provide a limited set of presentation-level customisation options. Thus, one can specify font type and size for textual information (class, attribute and operation names), set relationship lines to be rectilinear or oblique, and determine the visibility of attributes, operations and their features (e.g., whether public or private). Other presentational features, such as line style, are not available for change, most likely because they are deemed to be central to the visual conventions of the UML diagramming style.

Such limitations on what is, and is not, configurable at run-time results in an unnecessarily inflexible tool. It is inflexible because there may well be cases of useful visual modifications unanticipated by the original tool builders (e.g., to improve readability or to increase information content). Consider, for example, setting the line width of an inheritance link to correspond to the number of operations which are inherited by (passed down to) a child class. This breaks no known convention but might provide valuable information in certain circumstances.

Furthermore, if there are certain features that should not be changed in a particular context of use, constraint change permissions can be restricted by group-based ownerships. Changing line style might be available, say, to a project leader but not to team members. Changing the appearance or behaviour of a modelling tool is like changing the syntax or even semantics of a visual language. As with any language, such changes can be risky - confusion or misinterpretation can result [9]. But without change any modelling language, just like a natural language, is doomed to eventual obsolescence. By using run-time constraints, decisions about appearance can be fixed relative to a context of use without being fixed permanently. Indeed, a suitable constraint manager can report on violations of company conventions, say, without prohibiting their violation.

5 The Architecture of a Constraint-Based Modelling Tool Generator

To provide structured run-time customisation of the type described above, a meta-CASE tool must support:

- the specification of executable constraints, both embedded and run-time, applicable to the relationship, representation and presentation levels of description
- inclusion of design-time constraints as embedded constraints
- a configurable constraint manager, capable of changing constraints from hard to soft and of providing different forms of constraint-status reporting

We propose an architecture for the target modelling tool, including a software model component consisting of an augmented graph object and a linked view component which implements the visual appearance and interactive capabilities of the tool. In addition, the target tool will include set of run-time constraints and a constraint manager, responsible for handling the resolution of these constraints. Figure 2 illustrates the architecture of modelling tools produced by our proposed meta-CASE system.

Modelling tools are specified using a specification language for modelling techniques. A modelling tool specification includes two parts:

- base-level tool description: this defines the basic graph and its interpretation plus the user interface representation of the interpreted graph
- run-time constraints: a set of run-time constraints on the graph, its interpretation and its presentation. A tool generator takes a modelling tool specification and generates a modelling tool by producing graph and presentation components (augmented graph and graph view, respectively, in Figure 2) according to the base-level tool description and a set of run-time constraints according to the run-time constraint set.

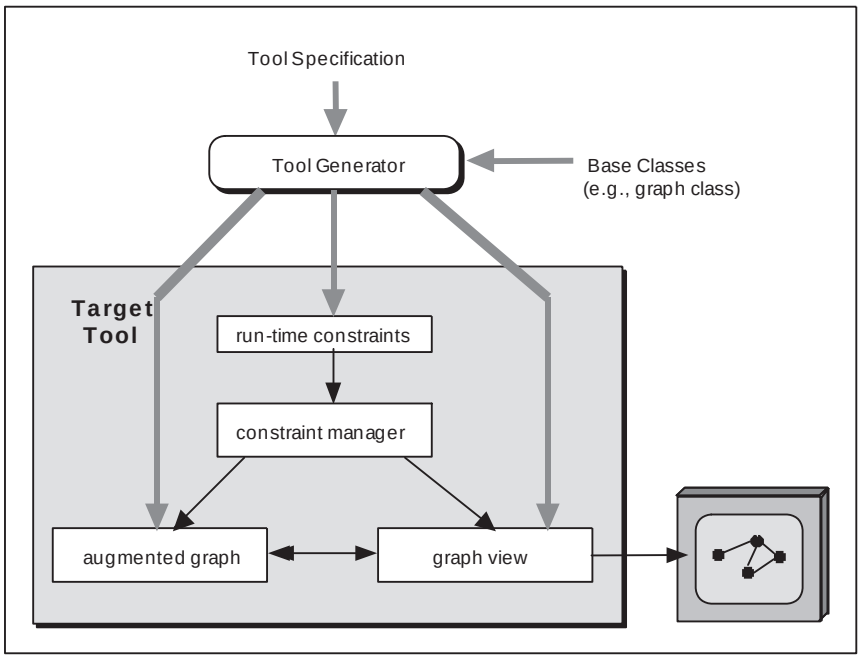

Figure 2: The Target Modelling Tool Architecture

5.1 A Prototype Tool Generator

Serrano and Welland made a first attempt at the design and implementation of a specification language and associated generator that separates tool syntax from run-time constraints. VCt [17] is a language to produce formal specifications of diagram-based modelling techniques. A number of complete specifications of modelling techniques have been produced with the language, including Data Flow diagrams (DFDs), state transition diagrams and Entity-Relationship diagrams. Serrano developed a prototype supporting system [15] that includes a compiler with a code generator, a generic tool to manage node and link diagrams that enforces constraints at run-time, and tailorable user interaction facilities.

This prototype system generates a complete design editor tool for the end-user, supporting the editing of a specified diagram type. The constraints are implemented as code fragments generated from the VCt specification and compiled into the code of the end-user diagram-editing tool.

The system exploits run-time diagram constraints, based on assertions that form part of the diagram specification, that are maintained by the diagram editor during execution. This notion of constraint is similar to that of integrity constraints in databases, i.e., properties, which must be maintained by the component objects, and operations which constitute the diagram editor. For example, one might specify that in a DFD there must be at least one external entity that provides input to the system. This condition can be asserted in the VCt language as follows:

```
C9:"There must be a least one
external entity which provides
input to the system"
∃e:ExternalEntity •
      e ∈ Externals (dfd) ∧
∃f:DataFlow • f ∈ Dataflows(dfd) ∧
      (source(f) = e)
```

ExternalEntity and DataFlow are declared as icons and connections, respectively, and source is a property of DataFlow that returns an icon. This assertion will be transformed into a constraint that must be satisfied by any diagram created using a diagram editor that meets the VCt specification. The editor will include code to ensure that this constraint is maintained during diagram construction by the

end-user.

A library of pre-defined functions is provided to extend the expressiveness of the VCt language and simplify writing predicates. A common constraint in many modelling techniques is that all elements of a diagram must be connected, so a function, which cannot be written directly in VCt, is provided. A function called UniqueName is also provided to traverse all diagram elements of a particular type to ensure that they all differently named, another very common constraint. This can be written as a straightforward predicate in VCt but occurs so frequently that it is useful to provide a function.

The process of producing a modelling tool with the system is done in two steps:

1. A given modelling technique is specified using the VCt language, including assertions that will eventually be transformed into run-time constraints,

2. Code is generated automatically to implement the modelling tool supporting the specified modelling technique, including code to maintain the constraints. Specifications for several modelling techniques have been produced and compiled [16]. As a more extensive test, Serrano produced a complete state transition diagram-based design tool using the prototype generation system. These preliminary findings establish the feasibility of the whole approach, viz., constraint-based specifications can be used to describe diagram-based modelling techniques and from these, design tools can be created automatically.

5.2 Developing a Tool Generator with a Constraint Manager

The prototype system generates embedded code for constraints directly from the VCt specification using templates for different types of constraints [15]. In order to implement the architecture shown in Figure 2, we need to:

- define an intermediate constraint specification language which can be interpreted by the constraint manager

- modify the tool generator so that it translates constraints from the formal notation of VCt into the new specification notation

- build a constraint manager which can be invoked to check constraints when specified events occur during diagram construction at run-time

- enhance the user interface to allow the end-user to control constraint execution and receive feedback from the constraint manager subsystem.

Currently, we are working on a Java-based model editor framework consisting of a graph editing framework (based on GEF [12]) augmented by domain enhancements and a prototype constraint manager with a user interface that presents constraint-satisfaction status in the form of critiquing-oriented reports and that permits control over constraint resolution policy and execution.

We have taken a small subset of VCt constraints, those involved with naming in DFDs (including naming constraints dependent on graph properties), and hand-translated them into an intermediate human-readable format in XML. These constraints are then transformed by the run-time editor into equivalent Java objects representing the instantiated constraints. The prototype, albeit limited in functionality, demonstrates the feasibility of configurable constraint resolution policy. We are now in the process of adding presentational constraints and automatic editor generation from VCt specifications.

6 Related Work

Commercial software modelling tools exhibit relatively limited customisation facilities. For example, Rational Rose, a popular set of tools for object-oriented modelling using UML, has interactive facilities for customising the appearance of diagram elements and for incorporating user-defined operations [11]. However, the basic semantics of diagrams cannot be inspected as a specification nor can the diagram structure or basic behaviour be modified. EiffelCase from ISE [5] offers facilities for building and displaying concurrently multiple views (viz., alternative presentation levels), but these have only limited configurability.

The Alvey-funded ECLIPSE project included the production of a design-editor [3] that could be tailored to support different software design notations. This system was based on a single generic

design editor driven by a set of tables generated from an informal specification language called GDL [19, 22]. A tool builder could specify a design notation using GDL that was then compiled to create the design-editor tables. The editor's behaviour could be changed by modifying the GDL description and recompiling the tables.

GDL is an ad hoc specification language that mixes design concepts with graphical representations of modelling techniques. Therefore, it is not possible to reason about specifications (e.g., to identify conflicting constraints). GDL also contains distracting detail about graphical representation and spatial constraints that is not fundamental for software modelling. The ECLIPSE DE constraint system is unsophisticated and does not allow for constraint classification or dynamic configuration at run-time.

After the ECLIPSE project, the basic components were re-engineered into a meta-CASE product called Toolbuilder [21, 2], now maintained by Lincoln Software. Tools are created by parameterising generic tool elements and composing the specialised elements to create a tailored design tool. The parameters are generated from a specification of the desired method. When generic tool components do not provide the required functionality, method specific procedures can be added using the interpreted language Easel.

More flexibility is offered by a metamodelling approach in which a metamodel framework is populated with the particular model components required for a desired diagramming tool. MetaEdit+ is a commercial model development environment that uses this approach [18]. Although there is an inspectable high-level description of the diagram (its model) there are no facilities for exploring the consequences of diagram design decisions in the run-time system. That is, a designer cannot explore diagram design alternatives without changing the underlying model and rebuilding the editor. Furthermore, a meta-modelling approach distributes diagram behaviour across model components, making it difficult to predict how the diagram will behave. A constraint-based architecture, in which declarative assertions about diagram semantics are transformed into run-time constraints satisfied by an interactive diagram modelling tool, could overcome the restrictions in specification and customisation encountered in current diagram development tools.

MetaView [7, 20] is a meta-CASE system that includes a constraint language, called ECL (Environment Constraint Language). MetaView distinguishes between conceptual information, the semantics of a design, and graphical information stored about the presentation of that design. ECL can be used to express both conceptual and graphical constraints. The conceptual constraints specified in ECL have similar expressive power to constraints defined in VCt, although ECL additionally includes constraints on aggregates, such as multi-level DFDs. There is no separation of the MetaView conceptual constraints into base representation (the "relationship level") and interpretation (the "representation level") constraints as in our system.

MetaView conceptual constraints are divided into consistency and completeness constraints. Consistency constraints are checked whenever a graphical operation is carried out that could change the semantics of the conceptual design information, stored in the specification database. Violation of a consistency constraint generates an error message, which must be acknowledged by the user before proceeding and without updating the specification database (i.e., the operation is cancelled). Completeness constraints are applied when the user explicitly requests a completeness check. Any violations of completeness constraints are reported to the user via a list of error messages. The MetaView approach links constraint type (consistency vs. completeness) with constraint effect (abort operation vs. report violation). We have separated these aspects of constraints - category and effect, treating them as orthogonal and offering a designer/tool user the option of choosing the constraint effect policy independent of the constraint category.

The UML-based design environment Argo [13] employs run-time constraints on graph and model properties as a means of offering design critiques. Argo does not, however, provide the same constraint mechanism at the presentation level.

7 Conclusions

We have argued that constraint-based customisation is superior to the rather ad hoc approach of current commercial modelling tools and, when incorporated into a meta-CASE generator, provides a

powerful means of realising flexibility with minimum implementational cost (i.e., no rebuilding necessary to regenerate constraints). By combining constraints with a basic tool specification, modelling techniques can be defined, promoting reuse and controlled tailoring and evolution of techniques.

A limited prototype tool and language have already been developed [10]. We are now engaged in re-engineering and enhancing this prototype prior to an evaluation of our method.

A number of questions remain unanswered:

- Can our language and architecture support a sufficiently rich range of basic modelling techniques and forms of customisation? We have done some work on specifying modelling techniques such as STNs, single-level DFDs and ER diagrams. However, we need to specify a wider range of modelling techniques and deal with some awkward problems like abstraction/explosion in multi-level diagrams such as DFDs.

- Can we support a rich enough variety of customisations using our constraint language? Will toolbuilders and software modellers find the constraint model usable and useful? We intend to tackle this problem from two angles. First, by investigating the use of existing tools we can identify potentially useful types of constraints to provide flexibility. Second, we shall carry out user testing of our prototype and its descendants to identify the viability of the constraint approach. These empirical investigations will include small-scale controlled experiments to explore specific usability questions and longer-term studies of use in realistic industrial settings.

- Constraint violations can be monitored and logged. Such a log forms a sort of ”design history” of tool use during diagram construction by end-users. Can such design histories prove useful in identifying problematic parts of a model and reusable modelling patterns?

- Can tool users customise constraints expressed in our language or is a more comprehensible representation needed in this context? Our present constraint language is formal, based on Z notation, and we need to investigate whether this is usable by software engineers or whether we need to provide alternative interactive specification techniques for constraint specification.

- Should the constraint manager itself be configurable? Our initial assumption is that modifiable constraints along with a fixed global constraint manager will offer the flexibility needed. However, we might consider whether the constraint manager itself should be configurable, e.g., to allow selection of alternative actions on constraint violation.

Acknowledgements

Dr Artur Serrano developed the VCt language and a prototype tool builder as part of his PhD work while studying at the University of Glasgow. He has made a large contribution to the approach described in this paper. The authors would also like to thank the anonymous reviewers for their helpful comments.

About the Authors

Phil Gray is a lecturer and Ray Welland a senior lecturer in the Computing Science Department of the University of Glasgow, Scotland.

References

[1] A. Alderson. Meta-CASE Technology. In *European Symposium on Software Development Environments and CASE Technology*, Konigswinter, Germany, 1991.

[2] A. Alderson, J.W. Cartmell, and A. Elliot. ToolBuilder: From CASE Tool Components to Method Engineeering. In *CoSET 99 Workshop*, May 1999.

[3] S. Beer, I. Sommerville, and R. Welland. The Design Editor. In F. Bott, editor, *ECLIPSE: An integrated project support environment*, pages 85–95. Peter Peregrinus, 1989.

[4] A. Borning. *Thinglab - A Constraint-Oriented Simulation Laboratory*. PhD thesis, Stanford University, 1979.

[5] EiffelCase: Graphical support for seamless, reversible O-O Development. Interactive

Software Engineering, 1999.
http://eiffel.com/products/case/page.html.

[6] B. Myers et al. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. *IEEE Computer*, 23(11):71–85, Nov 1990.

[7] P. Findeisen. The metaview system. Technical report, Dept. of Computing Science, University of Alberta, 1994.

[8] P. Gray and S. Draper. A Unified Concept of Style and its Place in User Interface Design. In *Proc. of HCI '96*, pages 49–62. Springer-Verlag, 1996.

[9] T. Green and M. Petre. Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, pages 131–174, 1996.

[10] Java Beans API Specification. Sun Microsystems.
http://java.sun.com/beans.

[11] Rational Rose 98. Rational Software Corporation, 1998.
http://www.rational.com/products/rose/.

[12] J. Robbins, D. Hilbert, and A. Gauthier. Gef: Graph editing framework. Technical report, Dept. of Information and Computer Science, University of California, Irvine, 1999. http://www.ics.uci.edu/pub/arch/gef/index.html.

[13] J.E. Robbins and D.F. Redmiles. Software architecture critics in the Argo design environment. *Knowledge-Based Systems*, 11:47–60, 1998.

[14] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[15] J.A. Serrano. *Automatic Generation of Software Design Tools Supporting Semantics of Modelling Techniques*. PhD thesis, University of Glasgow, 1997.

[16] J.A. Serrano and R. Welland. Complete VCt Specifications of Modelling Techniques. Technical Report TR-1997-24, Department of Computing Science, University of Glasgow, 1997.

[17] J.A. Serrano and R. Welland. VCt - A Formal Language for the Specification of Diagrammatic Modelling Techniques. *Information and Software Technology*, 40(9):463–474, 1998.

[18] K. Smolaner, K. Lyytinen, and V. Tahavanainen. Meta-Edit - A flexible graphical environment for methodology modelling. In *Proc CAiSE91*, 1991.

[19] I. Sommerville, R. Welland, and S. Beer. Describing Software Design Methods. *Computer Journal*, 30(2):128–133, 1990.

[20] P. G. Sorenson, P. S. Findeisen, and J.P. Tremblay. Supporting Viewpoints in Metaview. In *SIGSOFT 96 Workshop*, pages 237–241, 1996.

[21] Tool Builder. Lincoln Software.
http://www.lincolnsoftware.com.

[22] R. Welland, S. Beer, and I. Sommerville. Method Rule Checking in a Generic Editing System. *Software Engineering Journal*, 5(2):105–115, 1990.