Dressing AI in COTS Clothing

Robert M. Balzer
USC/Information Sciences Institute
balzer@isi.edu

Neil M. Goldman
USC/Information Sciences Institute
goldman@isi.edu

From: AAAI Technical Report WS-99-09. Compilation copyright © 1999, AAAI (www.aaai.org). All rights reserved.

Abstract

Although, the benefits of "domain specific" languages and development environments are widely recognized, constructing a design environment for a new domain remains a costly activity, requiring expertise in several areas of AI, software development, and the targeted domain. We've simplified this task by combining a design editor generator with a COTS product (Microsoft PowerPoint) that provides both the graphic middleware and end-user GUI for the generated design editors.

Introduction

Domain-specific languages & development environments are frequently proposed as a means to improve the productivity of designers [1]. Although prototypes of such languages and environments proliferate in conference proceedings, commercially viable examples remain rare. We believe that the reason for this is primarily the difficulty of *implementing*, not of designing, a high-quality design environment for a new domain.

There are two major parts of a domain-specific design environment for an engineering domain. The first is a graphic user interface that lets an engineer intuitively manipulate the objects constituting a design, create reusable sub-designs, and navigate within and between designs. The second is an integrated toolset that provides the engineer with feedback on a design – problems, metrics, scenario animations, etc.

We believe that the first portion – the GUI – requires only shallow knowledge of the application domain on the part of the environment builder. The second problem, although it may have graphical presentation aspects, relies on a much deeper understanding of the domain.

The ISI design editor generator addresses these two areas in disparate ways. It simplifies the GUI-building task by extending a high-quality commercial, but non-domain-specific, platform for constructing and presenting graphics – Microsoft PowerPoint – rather than some lower-level graphic library such as Motif or GUI constructors such as VisualWorks or Visual Basic. The generator's "specify by example" paradigm casts the creation of the GUI for a new domain as a graphical task in its own right, rather than a programming task. PowerPoint itself provides the preponderance of the design editing GUI, which is common across engineering domains.

The design environment generator provides a flexible runtime architecture for incorporating feedback programs (called *analyzers*) into the generated environments. These analyzers can be written in the programming language, and run on the machine, of the implementer's choosing. The communication protocols used by the analyzers and the design editor allow analyzers to be written using either batch-oriented or incremental algorithms. This flexibility should make it relatively easy to import preexisting domain-specific feedback programs into the generated design editor environments.

The analyzer-editor protocols also support common graphical presentation requirements of feedback, permitting the design editor to reflect analyzers' results directly onto a graphical design, rather than requiring an analyzer to provide its own GUI for that purpose.

Figure 1 shows the roles of the domain expert, the analysis programmers, and the GUI designer in producing a domain-specific design environment for engineers.

Domains and Designs

Common to numerous engineering domains is the "box-and-arrow" character of visual designs. Boxes represent *components* of a design artifact. Each component denotes an instance of some *component type* – resistors and capacitors, tasks and workers, cargoes and vehicles – the types used are highly domain specific. Each arrow represents a relationship between the components at the ends of the arrow. These relationships may be physical, temporal, or neither. A single design may reflect several different *relationship types* – such as control flow and data flow in a software algorithm. In most domains, not all instances of a given component type are identical. So the types are parameterized by *properties* — such as the capacity of a storage tank or the power of a lens. Like components, relationships may also have properties – such as the gauge of a wire or the delay of a communications link. We currently support properties with boolean, integer, real, and string values, as well as with values from domain-specific enumerated types. A property value may consist of a single value from one of these types or a set of values.

The *units* of a design are the component and relationship instances in the design. Knowing the unit types of a domain and the properties of each constitutes the shallow syntactic knowledge of the domain. By itself, it is not sufficient to produce a semantically meaningful, much less useful, design.

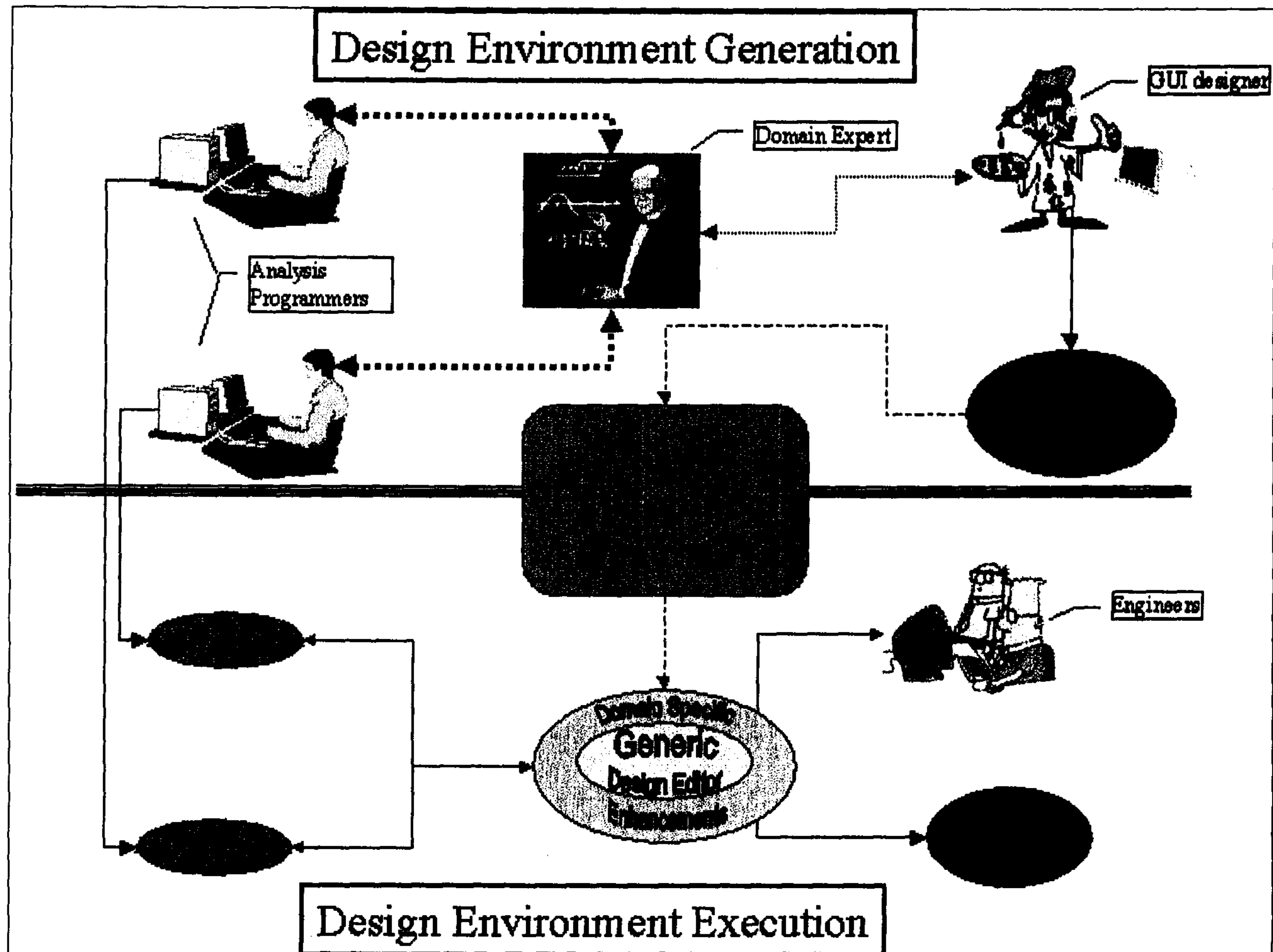

Figure 1. Design environment generation

Nevertheless, this shallow knowledge is significant because *this is the level of information exhibited in graphic designs.* This simply reflects the fact that this level of representation is sufficient for two crucial purposes:

- Engineers (or software) with a deeper understanding of the domain can *derive* the information they need from it. It thus serves as the basis for analysis and communication between engineers.
- Other people (or software) can construct artifacts from it (i.e. implement the specified design) *without the need for a deeper understanding of the domain.*

Our work focuses on leveraging the central role of these shallow domain models within in a design environment. Our contributions are:

1. Generate a domain-specific design editor for a domain without any traditional programming [see Generating Design Editors section]
2. Provide a framework for analysis programs to track an evolving design and provide feedback. [see Analyzers and Analyses section]
3. Generate the domain model for these analysis programs [see Generating Design Editors for New Domains section]

Analyzers and Analyses

Although graphical designs are often used solely for their value for human visualization and communication, they become more valuable if software tools can also provide analyses and/or implementations of a design. Informally, we consider an analysis to be any body of information derived from a design. Examples of analyses are:

- Design correctness feedback
- Cost and performance analyses
- Automatically generated implementations of software designs
- Animating a usage scenario on a design

Each domain has its own idiosyncratic analyses, whose requirements for design data, synchronization, and feedback mechanisms may vary substantially. To accommodate these variations the design environment architecture allows analyzers to be independent

components that communicate with the editor through an object-oriented protocol for exchanging design information and analysis feedback. The design editor provides an analyzer with incremental updates to the design state. An analyzer may also query the editor to find out about particular aspects of the design state. This allows a variety of implementation techniques to be used in analyzers.

An analysis may be parameterized. The parameters of an analysis are just like the properties of a design unit, with one exception. An analysis may be "focussed". What this means is that it has a parameter consisting of a set of units from the design being analyzed. An analyzer will typically use this focus parameter to restrict its analysis to the portion of the design designated by the focus set.

Analyzers execute as separate processes, possibly not even on the same machine as the design editor itself. The relative independence of analyzers means that an analyzer could implement its own GUI for presenting analysis results to a designer. However, to simplify the implementation of analyzers, and provide for graphical presentation of feedback on the design itself, analyzers *may* make use of a predefined reporting mechanism in the analyzer-editor protocol.

An analyzer may send the editor an *analysis* consisting of one or more *results*. Each result consists of a textual *explanation* together with a (possibly empty) set of *markups*. The markups provide graphic feedback to augment the explanation. Each markup can specify:

- that a unit be highlighted
- that a unit be hidden
- that a component port or arrow terminus be labeled with specified text.

For example, a report might have the explanation "Only one input is allowed at the control port of a thermostat." The accompanying markups might call for highlighting two arrows terminated at the same control port of a thermostat, and labeling that port with the text "too many inputs".

We divide analyses into two categories: *snapshot* analyses and incremental analyses. An incremental analyzer that uses the report/markup mechanism for presenting feedback is expected to update the analysis each time that it receives an update to the design state.

Updates to the design state are actually grouped into transactions in the editor-analyzer protocol. Incremental analysis updates are expected to follow each transaction. A designer might select several components through the editor GUI and delete them all with a single command. The editor groups the deletions into a single transaction to report to analyzers. This avoids the need to report analysis updates relative to ephemeral states that are meaningless to the designer.

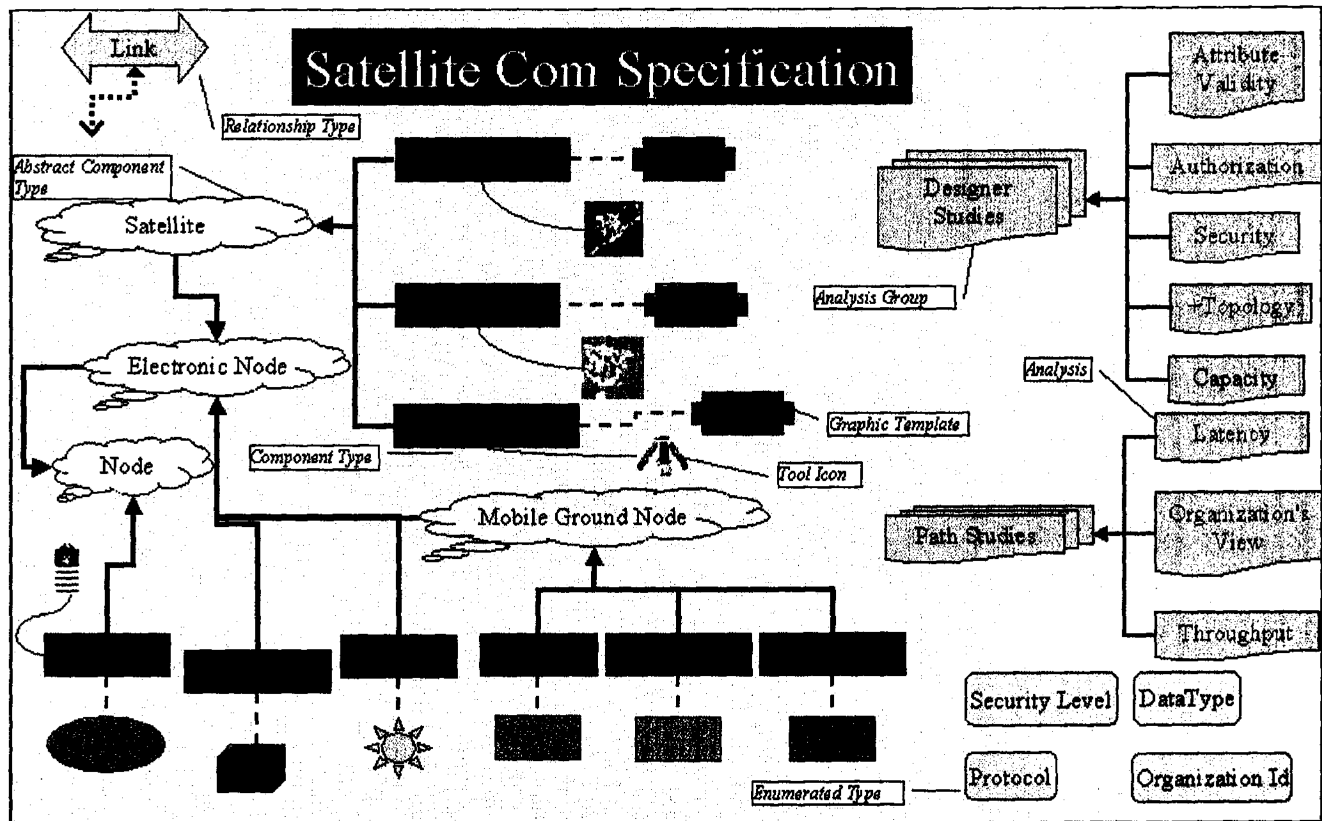

Figure 2 Domain Specification – satellite communication

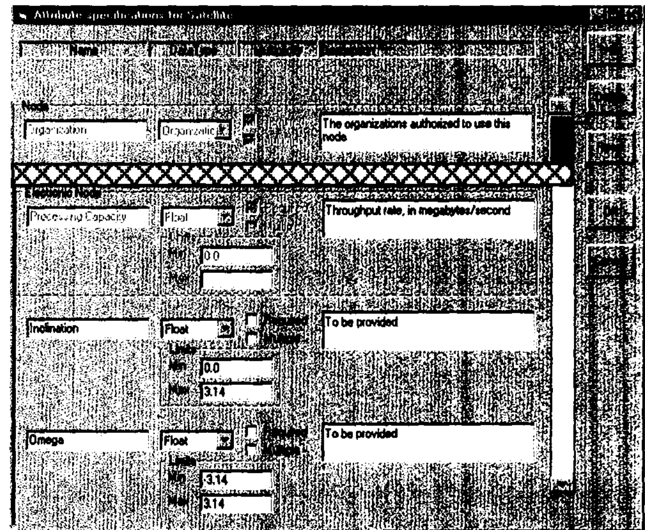

Figure 3 Property specification dialog

Specifying New Domains and Generating Design Editors for them

The GUI described below in the Design Editor GUI section contains no novel features. We wish to reiterate that there is only a superficial understanding of the domain represented in the GUI itself, excluding the *content* of the analysis results. The novelty comes from two sources, the first of which is the means used to generate that GUI. The second, extending a widely used COTS product, is discussed in the next section.

The "Satellite Communications" GUI was generated with no traditional programming. Its specification, created through another graphic interface, is shown in Figure 2. The (green) rectangles labeled "Comsat", "Sensor", "User", etc. determine the domain's component types. The cross shapes attached to them by dashed connections are their *graphic templates*. This determines the appearance of an instance of the type when an engineer instantiates it in a design. The GUI designer either chooses a graphic template from a large library of shapes, or may import an image, in any of a variety of image formats, as a graphic template. The GUI designer tailors the template's color, border, and label text in this graphic domain specification.

A type specified may be connected (via a curved solid connector) to an image that serves as the *tool icon* for the type in the generated domain toolbar. Tool icons, like graphic templates, may be selected from a shape library or use an imported image. If no tool icon is specified, a scaled version of the graphic template is used as the tool icon.

The (gold) arrow shape labeled "Link" provides the sole relationship type in this domain. The dashed, double-headed arrow attached to it is the graphic template for the "Link" relationship type. The GUI designer tailors the color, dashing and arrowhead styles of a relationship template in the graphic domain specification just as he tailors component type templates.

Single-inheritance hierarchies of unit types can be specified by placing abstract types, such as "Satellite", in the design. Properties can be associated with either abstract or unit types. Property definitions are entered through a dialog like the one in Figure 3. A specification consists of a name, a type selected from a drop-down list, optional upper/lower bounds for numeric types, required/multiple indications, and a textual explanation. The explanation will appear in a small pop-up window when the designer hovers the mouse on the "tab" for that property in a property-editing dialog.

Any unit or relationship type may have initial property values specified through a property-editing dialog, identical to the ones used by designers. The default values are assigned when new instances of the type are created.

Figure 2 contains the specification of two global root analysis groups, "Designer Studies" and "Path Studies", and eight analyses. The color and styling of the border of an analysis specify the means used to highlight components and relationships directed from markups in the feedback from the corresponding analyzers. Analogously, the text characteristics – font, face, size, color – of the label of an analysis specify the textual characteristics of any on-design markup text found in feedback from the analysis.

Generating Design Editors for New Domains

The specification-by-example editor is little more than a domain-specific editing environment specified with its own (partially bootstrapped) graphic domain specification for the "domain-definition domain". A PowerPoint presentation file created by editing a design in the domain-definition domain serves as the specification for a new domain. Currently, the file name itself serves as the new domain's name. When a designer begins editing a design for a domain D, D's graphic specification is loaded in an invisible, read-only, mode into PowerPoint The design editor then extends PowerPoint's GUI by interpreting the content of that graphic domain specification.

We have implemented two "analyses" for the domain-definition domain. The first reports various errors such as unnamed types, circular inheritance, types without templates, etc. The second "analysis" is a generator that produces an ASCII file containing definitions (in CommonLisp) for classes that correspond to those defined in the domain definition. A domain-independent CommonLisp module provides a mapping between this OO model and the editor-analyzer protocol. CommonLisp analyzers can then be implemented for this domain by programmers without any knowledge of DCOM and with all of the classes of that domain suitably defined.

Design Editor GUI

The central component of the design editor is its GUI. The editor's *GUI* provides the interactive user with means to load/save designs, navigate within designs, create/delete/copy components and connectors, view and modify properties of components and connectors, and request analyses.

Figure 4 below is a screen shot of an editor generated for a "satellite communications" domain. Everything in the figure is part of the GUI with the exception of the callouts highlighting specific elements.

Readers familiar with Microsoft PowerPoint will immediately recognize many elements from that product's GUI in this figure. This is discussed in detail in the Advantages of Extending PowerPoint section. Here we focus on the domain-specific aspects of the GUI.

In the central canvas is the design of a "satellite communications" configuration. The various labeled shapes represent instances of satellites, terminals, switches, processors, and users – the component types of the domain. They are connected by arrows representing communication links – the only relationship type used in this domain.

The designer created these design units through unit creation tools on the domain toolbar, seen near the upper right of the figure. To the immediate left of these tools is a drop-down list box displaying the name of the domain ("Satellite Com"). When a designer starts a new design, this box allows him to choose a domain. This triggers creation and display of the appropriate domain toolbar. Manipulation of units on the canvas – positioning, resizing, selecting, attaching/detaching links – is carried out through conventional mouse gestures and/or keyboard shortcuts.

The window displays a list of reports. In this example, there was just one report. Its explanation reads "User U3 is directly connected to user U2." When the designer selects a report, its associated markup instructions are carried out. Their effect is reversed if the report is deselected, or the analysis window is closed. In this case, the only markup instruction called for highlighting the communication link between U2 and U3. That is why that link has an appearance (a thin red arrow) different from the others.

Property values are viewed and assigned through dialogs, displayed on demand from the unit context menus.

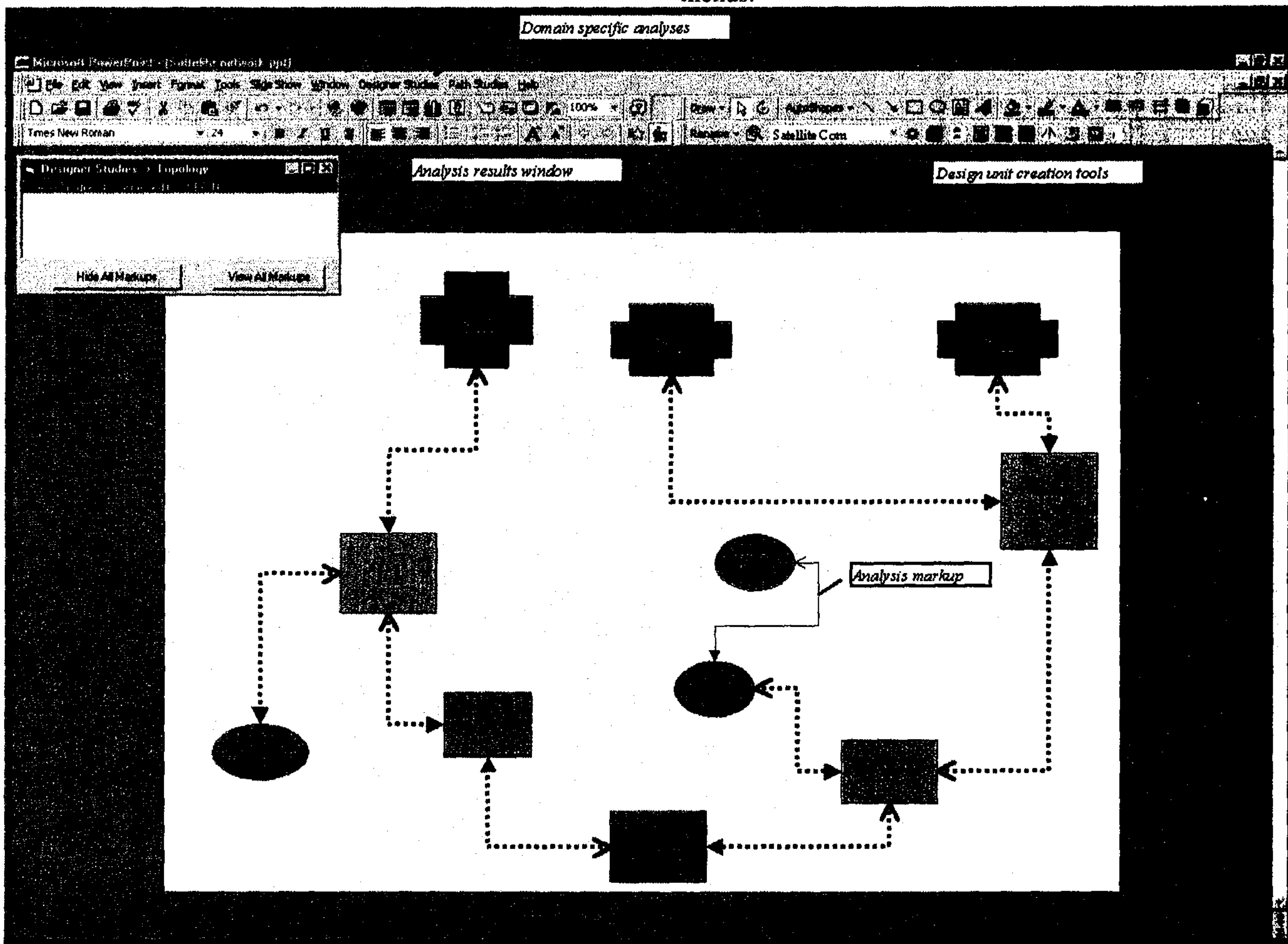

Figure 4. Design editing GUI – satellite communications domain

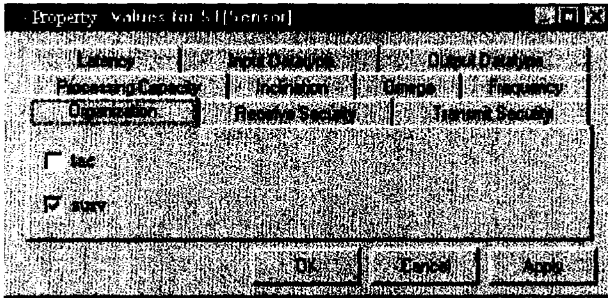

Figure 5. Property value dialog

Figure 5 exhibits the dialog for a sensor satellite. The dialog contains a "tab" for each property associated with the type in the domain specification. The details of a tab depend on the value type of its property and on of the domain specification. Identical dialogs are used to gather the parameter values for parameterized analyses.

Implementing the Design Environment with a COTS product

The design editor is implemented as an extension to Microsoft PowerPoint, programmed in Visual Basic. Technically, this extension is a COM [] server that receives "events" as the user modifies a design. The same module acts as a COM client of PowerPoint enabling it to navigate through a design and to paint analysis feedback directly onto the design. For efficiency reasons, this module runs as an "in-process" server. This means it is actually part of the PowerPoint process itself. Method calls are extremely fast when both client and server are part of a single operating system process. Greater efficiency could be achieved by implementing the extension in C++, but the performance of the Visual Basic code has been acceptable to us to date.

Design editor - analyzer protocol

When an analyzer process starts, it registers its interest in one or more domains, and registers as the provider of one or more analyses. As designs are loaded into the design editor, or modified, the design editor receives events from PowerPoint, interprets those events in terms of changes relevant to analyzers, and notifies registered analyzers. This portion of the editor, which we call the Design Monitor, is described in further detail in paragraph 0.

When a designer requests an analysis and provides its parameters, the design editor notifies the registered analysis provider. That analyzer is subsequently expected to send the design editor an analysis. The design editor then presents the analysis to the designer.

Every update sent to an analyzer is marked with a monotonically increasing transaction count. When an analyzer reports an analysis, it includes the transaction count at which the analysis was computed. Any visible analysis based on a non-current design state is visibly marked as out-of-date by coloring the background of its report window.

When an analyzer has provided an incremental analysis, it is expected to update the analysis each time it receives a design update from the editor. When the designer closes an incremental analysis, the analyzer is notified and ceases to transmit updates.

The protocol allows an incremental analysis to be updated either by total replacement or by selective deletion and addition of reports

Communication between the design editor and analyzers takes place via distributed COM (DCOM). The rationale for choosing DCOM over, say, CORBA, to implement communication between the design editor and analyzers is only that Visual Basic, the language in which we implemented the design editor, trivializes the implementation of DCOM clients and servers. The fact that PowerPoint itself exposes (D)COM interfaces is *not* a factor, because currently the design editor does not pass analyzers direct references to any PowerPoint objects. .

Although we have not done so, it would be reasonable to further categorize analyses as synchronous vs. asynchronous. Synchronous analyzers could use a simpler protocol (no need for transaction counts) and be allowed to run as "in-process" servers for high performance.

Advantages of Extending PowerPoint

PowerPoint is marketed as, and known to most of its users as, a presentation graphics editor. As such, it is viewed as an interactive editor of *presentations* consisting of multiple *slides.* However, it is also a *high level* graphic server, permitting independently written modules to read and update almost any aspect of its state and invoke numerous methods through an object-oriented COM interface. But what does PowerPoint offer that is missing from traditional "visual interface" authoring tools?

Primarily, PowerPoint offers a highly functional GUI for interactively designing presentation graphics. Virtually every part of that GUI is useful, *without modification,* as part of our design editor. This includes:

- Scrolling, zooming, scaling, multi-slide designs
- Loading/Saving/AutoSaving designs, multiple windows, multiple views.
- Object deletion, selection, grouping, cut/copy/paste, and text formatting.
- Object positioning, alignment, rotation, reflection, resizing, graphic formatting.
- *Connectors* – self-routing lines/arrows whose ends attach to other objects, and adjust automatically to repositioning and resizing.

We emphasize that it is not simply the fact that PowerPoint has a library with methods for accomplishing these operations, but that it has a functional GUI that allows the designer to invoke them conveniently. If one thinks of an engineering design as a specialized PowerPoint presentation, it is not surprising that we have found no reason to *remove* any of PowerPoint's standard GUI. For example, any graphic object created through conventional PowerPoint tools may be placed in a design. Such *annotations* will persist with the design but will be invisible to analyzers – just as comments in programming languages are invisible to compilers.

One other feature of PowerPoint, though not part of its GUI, has also leveraged our implementation. PowerPoint allows arbitrary information to be associated with presentations, slides, and graphic objects in the form of string-valued tags. This is sufficient for the design editor's needs to store its own non-graphic design information, such as the property values that a designer has assigned to a unit. PowerPoint ensures that this information persists as part of the saved presentation document – no additional persistence mechanism had to be implemented for these extensions.

Finally, we note that the PowerPoint GUI is *already familiar* to many engineers, who use PowerPoint to present designs to clients and other engineers. In fact, some of them have commented that they have existing PowerPoint presentations they would like to *import* into our design editor.

Disadvantages of Extending PowerPoint

We should not give the impression that extending PowerPoint's GUI provides the same flexibility as building a hand-tailored design editor GUI.

The biggest impediment was the lack of "event" notifications in PowerPoint. Most of the design editor's activity must be triggered by some event in the GUI – or, more specifically, by a state change initiated from some GUI event. For our own GUI extensions (such as our dialog for editing unit attributes) there was no problem providing suitable notifications to the editor. However, detecting events initiated through the native PowerPoint GUI was a serious problem. Although a COM interface could make relevant events available, the interface implemented by PowerPoint97 *does not*. We developed a Design Monitor that employed two mechanisms to overcome this limitation.

The menu items and control buttons in the PowerPoint GUI are objects in the documented model. We found a way to replace them with equivalent ones whose reactions invoked our own code, which internally synchronously invokes the original reaction. The mechanism is obscure, but relies only on documented operations and is fully general. For menu items, this method works independent of whether the invocation is by mouse or by keyboard shortcut.

However, "wrapping" the action associated with a command does not always provide an efficient means to determine the design-relevant events performed by the action. An extreme example is the "Undo" command. Although we may have control both before and after PowerPoint executes that action, we have no effective means, short of a complete comparison of before and after states, to determine the relevant state changes. The best we can do is simply remove such tools from the GUI, which is trivial. However, removal is undesirable, because the tool provides useful functionality for design editing, just as it does in editing presentation graphics. We have not found a satisfactory solution.

Events initiated by mouse clicks and motion within a design window were far more problematic. PowerPoint provides its extenders with no insight into those events or, more interestingly, into the changes to its state that result from handling them. Like any Windows program, mouse events are communicated to PowerPoint by the operating system through a message queue. Like other Windows programs, PowerPoint often responds to the lowest level mouse events by placing other, higher-level, events into its own message queue. Mechanisms independent of PowerPoint allow us to monitor messages being removed from this queue. Based on observations from a "message spy" program, we have developed ad-hoc rules to determine localized bounds (generally the currently "selected" units) for what *may* have changed in a design. We can then efficiently determine what design-relevant changes *actually* occurred by comparing our cached old state with PowerPoint's current state within the affected locale. The fact that we are not concerned with most graphic details speeds up this comparison significantly.

Visio is a commercial product with many similarities to PowerPoint, including the mechanisms it uses to provide extensibility. Since Visio provides extensive visibility into its state-change events, it might provide a better technical fit to our needs.

Version considerations

Using a COTS product as a system component makes version upgrade concerns more significant than is the case with conventional runtime library components. How will a new version of PowerPoint impact the design editor? Because our designs and domain definitions are fully standard presentations, the new version will certainly automate any file format changes required to make them work. All our code that relies on the advertised (D)COM object model should require no change, because numerous other third party PowerPoint extensions rely on the same model. Existing menu items and controls in PowerPoint's GUI might be removed, relocated, or renamed, but adapting to those changes would be trivial. New controls or menu items might

appear, but our ability to wrap their actions simplifies dealing with them. However, because our rules for interpreting the significance of messages in the message queue are based only on *observation of the current version*, there is reason to expect they might have to be revised in potentially non-trivial ways.

Where is the AI?

The AI used in this effort is hidden behind the COTS GUI. This insight is the inspiration behind the title chosen for this paper.

It resides in two places. The first is in the generators used to automatically build the domain-specific design editor and analyzer domain model from the shallow domain specification. These generators operate during the construction of the domain-specific design editor and analysis framework to automate the construction of the domain-specific design environment within the host COTS product. Thus, only the results of this automation are visible during the execution of the generated design editor.

The second place that AI resides is in the domain specific analyzers employed by the generated design editor. While these analyzers could have been written in any language, as described in the Analyzers and Analyses section, we chose to write them for the domains that we've created to date in AP5 [8]. AP5 extends Common Lisp with an active database that supports inferencing, constraints, and triggers. These features allow us to define rule-based analyzers that are largely declarative.

Related Research

Much recent work in graphical editor generation [2,3,4,5] has centered on the use of graph formalisms for specifying a class of diagrams and layout constraints. Such grammars are difficult to compose and, we suspect, will find their use primarily in the *analysis*, not in the *generation*, of designs. This has been the experience with textual language grammars, which are almost universally used in the *parsing* of programs, but have shown little leverage (despite considerable effort at *syntax-directed* editing) in aiding source-code editing.

VisPro [6] and DoME [7] demonstrate that the construction and editing portion of a design environment can be specified far more concisely (and graphically), by someone with no knowledge of graph grammar formalisms. Correctness and semantics (in our view, two forms of analysis) are specified separately.

Our design environment generator extends this work in two areas. We have shown that modern COTS products provide a readily extensible platform for the implementation of a generated editor, saving considerable programming effort and providing an enormous body of valuable *domain-independent* GUI capability at no development cost. Careful separation of *editing* concerns from *analysis* concerns allows analysis algorithms (including correctness analysis) to be implemented independent of the editor. Analyzers access a design through an abstract syntax derived from the domain specification and provide abstract feedback that can be presented by the editor.

References

[1] *The Conference on Domain-Specific Languages, Proceedings*, The USENIX Association, Santa Barbara, Ca. October, 1997. http://www.usenix.org/publications/library/proceedings/dsl97/index.html

[2] M. Minas and G. Viehstaedt, "*DiaGen: A Generator forDiagram Editors Providing Direct Manipulation and Execution of Diagrams*", 1995 IEEE Symposium on Visual Languages, Darmstadt, Germany. IEEE Computer Society, pp.203-210.

[3] J. Rekers and A. Schürr, "*A Graph Grammar Approach to Graphical Parsing*", 1995 IEEE Symposium on Visual Languages, Darmstadt, Germany. IEEE Computer Society, pp.195-202.

[4] D. Lewicki and G. Fisher, "*VisiTile – A Visual Language Development Toolkitl*", 1996 IEEE Symposium on Visual Languages, Boulder, Colorado. IEEE Computer Society, pp.114-121.

[5] D-Q. Zhang and K. Zhang, "*Reserved Graph Grammar: A Specification Tool for Diagrammatic VPLs*", 1997 IEEE Symposium on Visual Languages, Capri, Italy. IEEE Computer Society, pp.284-291.

[6] D-Q. Zhang and K. Zhang, "*VisPro: A Visual Language Generation Toolset*", 1998 IEEE Symposium on Visual Languages, Halifax, Canada. IEEE Computer Society, pp.195-202.

[7] "*Domain Modeling Environment*" (DoME) http://www.htc.honeywell.com/dome.

[8] D. Cohen and N. Campbell. "Automating Relational Operations on Data Structures." IEEE Software, pages 53-60, May 1993.