Tackling the Adoption Problem of Domain-Specific Visual Languages

Holger M. Kienle, Anke Weber, Jens Jahnke, and Hausi A. Müller
University of Victoria, Canada
{kienle, anke, jens, hausi}@cs.uvic.ca

“And the plane will pretty much fly on its own. After a few commands are given to a Windows-based navigation program, the eye will pilot itself using a global positioning system. [Maj. John Cane] added, ‚If a Marine can use (Microsoft) Word, he can get this plane to fly.’”
– [Wir02]

1 Introduction

Research tools often fail to be adopted in industrial settings; both domain-specific languages (DSLs) and domain-specific visual languages (DSVLs) are no exceptions [Cit96]. Potential reasons why most of these tools remain lab orphans are, for example, users’ unfamiliarity, difficult installation, non-intuitive user interface, steep learning curve, and poor interoperability with existing development tools and practices. Ironically, while DSVLs can offer many potential benefits to the user compared to textual approaches, they introduce additional adoption challenges. For example, DSVLs break developer’s existing tool infrastructure and require developers to master unfamiliar concepts and paradigms. These challenges are often too demanding for users and DSVLs are not adopted, despite their potential benefits.

A textual DSL allows developers to reuse part of their existing tool infrastructure and to apply customizations to accommodate the DSL. For example, a favorite (programmable) text editor such as emacs is an important productivity tool for developers. A textual DSL can easily offer customizations such as a new emacs major/minor mode or error message parsing. Many paradigms that developers know from general purpose programming languages carry over to textual DSLs. Interactive textual editing, for example, has universally known concepts: cursor, cursor movements, marking/copying/inserting of text, etc. Developers know how to deal with error messages and how to navigate to the line and column that corresponds to the message (automatically). A similar argument can be made for debugging.

In contrast, the use of DSVLs often forces developers to abandon parts of their existing tool infrastructure in addition to learning new paradigms. Often, a productivity tool has been used for years and considerable knowledge and customizations have been built up over that time. Understandably, developers are reluctant to adopt a new DSVL (whose impact they cannot yet assess) if it means that they have to abandon their productivity tools. Advanced graphical editing lacks universal paradigms and not many developers are intimately familiar with it.\(^1\) The same holds for error message handling and graphical debugging.

\(^1\)At the 1994 IEEE Symposium on Visual Languages a Visual Language Comparison was conducted with five participating languages. It is reported that “more than one observer has noted that the various visual programs are
A DSVL’s graphical paradigm, which is visible at its GUI, is the most distinguishing feature compared to textual DSLs. While this enables DSVLs to offer a more intuitive domain representation compared to textual representations, their (typically idiosyncratic) GUIs pose the primary adoption challenge.

How can DSVLs offer GUIs that are familiar to users? We propose Adoption-Centric Tool Development (ACTD) as a software development approach that leverages widely-used, shrink-wrapped office tools by building DSVLs on top of them. Office tools such as Microsoft Visio and PowerPoint already offer general functionality that is expected of visual languages. For example, visual elements can be created and deleted; manipulated; connected; copied and pasted; and saved and loaded. Developers are already intimately familiar with this functionality because they use them on a daily basis to produce pictures for documentation and presentations.

Both Visio and PowerPoint are highly programmable and customizable. Thus, domain-specific behavior (e.g., the particular look of visual elements and how they are allowed to interact) can be crafted on top of the existing, general functionality.

There are related approaches that leverage shrink-wrapped software for DSVL construction [GB99] [Spi01] [Wil01]. However, this research typically emphasizes the benefits for the developer, but fails to address the potential benefits for the user and tool adoption issues.

We illustrate the ACTD approach with a case study of a DSVL in the embedded micro-controllers domain, called microSynergy. We describe three different implementation approaches that we have pursued, and discuss their benefits and drawbacks. The first two approaches use techniques that do not follow ACTD: DSVL generation with DiaGen, and DSVL construction by extending the MonarchGraph framework. Both approaches are typical examples of how DSVL are implemented. In the last approach, we discuss how to build microSynergy on top of Microsoft Visio to leverage existing infrastructure familiar to many users.

The paper is organized as follows. Section 2 gives a brief introduction to the microSynergy DSVL. Section 3 discusses the development approaches that we have pursued. Section 4 briefly summarizes our experiences and draws some conclusions.

2 The microSynergy DSVL

Distributed embedded applications are often developed from scratch and implemented using low-level programming languages like assembler or C. The software is often tightly coupled with specific hardware architectures, allowing little reuse and hindering interoperability. Furthermore, these applications are often more complex than the development of centralized control programs because of their inherent concurrency and event-based nature. This results in applications that are often only maintainable by programming experts who have an understanding of both the target hardware and the application domain.

microSynergy [Jah01] is a graphical development environment for the rapid construction of distributed embedded micro-controller software (cf. Figures 2 and 3). It allows to visually specify the interaction logic for communication among multiple micro-controllers with Specification and Description Language (SDL) finite state machines (FSMs). microSynergy research has been carried out in tight collaboration with Intec Automation Inc., a company in the area of embedded systems [Int].

SDL has been developed for the specification of complex, event-driven, real-time, and interactive applications involving many concurrent activities that communicate using discrete not easy to read, despite the alleged advantages of visual expression. This is a problem with visual languages; there is not yet enough shared learning about what computational icons mean” [Han94].
signals. It has a standardized graphical representation with entities such as blocks, processes, procedures, signal and type declarations, inputs, outputs, conditions, variables, states, and transitions. SDL models FSMs that run in parallel, independent of each other and communicate with asynchronous signals. An FSM consists of states (represented as rectangles with round corners) and transitions (represented by directed arcs); see Figure 1 for an example. The initial starting state is marked by an oval.

We chose a visual modeling paradigm to ease the programming task for engineers that are familiar with the (graphical) SDL formalism but have no knowledge of general-purpose programming languages. microSynergy’s visual programming approach has the benefit that engineers manipulate graphical SDL objects with which they are already familiar. Since engineers operate in their familiar domain, we can expect the work product to be more easily understood and maintained in comparison to other representations that lack *closeness of mapping*\(^2\). Furthermore, training costs are reduced and acceptance barriers are lowered.

### 3 Implementation Approaches

There is a large variety of approaches to implement DSLs in general and DSVLs in particular [vDKV00] [EJ01] [Spi01]. To implement microSynergy, we experimented with three approaches. Our first approach utilizes a diagram editor generator (DiaGen), the second approach

\(^2\)The concept of closeness of mapping is discussed by Green and Petre in the context of visual languages. They offer the following rationalization: “The closer the programming world is to the problem world, the easier the problem-solving ought to be. Ideally, the problem entities in the user’s task domain could be mapped directly onto task-specific program entities, and operations on those problem entities would likewise be mapped directly onto program operations” [GP96, Section 5.2].
is based on a graph data visualization framework written in Java (MonarchGraph), and the third approach is based on Microsoft Visio.

Wile calls the last approach *COTS-based* because it employs shrink-wrapped products to build DSVLs [Wil01]. He also shows how a (toy) DSVL that models a “satellite ground controller” domain can be implemented with Microsoft PowerPoint as well as Microsoft Access.

### 3.1 Generation with DiaGen

In an effort to minimize the time and effort of development we used DiaGen (Diagram Editor Generator) [MK99] [Dia]. It allows developers to quickly design and build graphical editors by providing (1) generic functionality for building and analyzing diagrams, and (2) generation of Java source code. The core components of a graphical editor can rapidly be generated by:

1. Creating a specification file (based on graph grammars).
2. Running DiaGen on the specification file to obtain Java code.
3. Editing the set of largely complete Java source files created by DiaGen.
4. Compiling the Java sources into Java classes.

DiaGen’s specification language allows rapid development of graphical components and relationships among any of the graphical diagram components. Graphical interrelationships can be checked and validated. DiaGen also provides layout managers for automatic repositioning of graphical elements.

#### 3.1.1 Experiences

DiaGen generates Java source code from the specification file. In our case, the ratio of generated Java code from the specification is about 20:1 (in number of lines). Unfortunately, we were forced to make changes to the generated code. Only very few of these changes were necessary, but they required a deeper understanding of the DiaGen framework.

Java sources generated by DiaGen dramatically speeded up the development. It took about a week to finish an editor prototype with basic editing functionality (i.e., placing, connecting, and moving of SDL entities). However, our team ran into serious difficulties when we tried to extend the generated editor with additional functionality "under the hood," for instance, introspection of existing embedded controller networks, automatic updating of the internal language model and generation of embedded code. We found that the internally generated abstract syntax graph implementation is quite hard to understand and modify consistently. Moreover, it was even more difficult to maintain consistency with our internal extensions each time we modified and compiled the DiaGen specification file. This problem occurred due to the generative nature of DiaGen: almost each newly generated code broke the integration with our extensions. The problem became even more dramatic with new releases of the DiaGen compiler toolkit.

Moreover, the DiaGen framework has a steep learning curve. In order to understand and develop specifications, one has to be familiar with hyper graph grammars and DiaGen’s style of expressing them. Also, the general editor framework has be to extended and customized (typically by subclassing) to obtain a domain-aware editor. Finding the—often undocumented—extension points (“hooks”) and understanding the extension patterns took time. It took an experienced DiaGen developer at least three days to train a new programmer in undertaking basic maintenance tasks.
In summary, we can report that DiaGen has served well for rapidly generating editor prototypes for our DSVL. However, extending and customizing these prototypes with additional functionality and subsequent maintenance tasks posed extreme software engineering challenges. Therefore we decided to move our tool development method from generative reuse to component-oriented reuse.

3.2 Framework Instantiation with MonarchGraph

MonarchGraph [MG] is a commercial Java GUI component framework for visualizing graph data structures. Domain-specific data can be visualized and edited by supplying an implementation of a data model class (via subclassing). Furthermore, the library can be extended by providing custom node and link views, diagram layout algorithms, and user interface plug-ins. Figure 3 shows a screenshot of microSynergy built with MonarchGraph version 1.1.

From a software engineering perspective, maintenance activities became much easier after we moved microSynergy to a component-based architecture (using MonarchGraph). New developers were able to understand the new architecture within one hour. A few technical difficulties (particularly with the MonarchGraph event model) could be solved or "worked around" with relatively little effort.

However, when we demonstrated the new version of microSynergy to our industrial partners...
at Intec, we were pointed to a completely separate category of issues with regard to usability. Some of these issues were related to our DSVL. For example, Intec criticized that there was no mechanism for the user to reuse previously developed communication patterns. Still, many issues were raised in regard to the so-called "counter-intuitiveness" of user interactions. We solved the first issue by extending our DSVL with a templates concept, which are out of the scope of this paper. However, the second ("tool adoption") issue was harder to grasp and we decided to conduct a more formal study to detect and categorize the usability problems in more detail [dMJ02].

3.2.1 Usability Study

The usability study was based on the following requirements: (1) microSynergy should be a general purpose tool that can be adapted to meet the requirements of many different users in many settings and (2) non-software engineers but computer literate engineers who had never programmed before should understand the visual modeling paradigm and use microSynergy with a flat learning curve.

For the formal evaluation of our prototype, we interviewed five subjects with different ranges of experience in software engineering. According to Nielsen, five subjects are sufficient to discover 75% of usability problems [DFAB98]. We asked the subjects a series of questions about their prior knowledge of software engineering, distributed programming, and SDL. Based on the subjects’ answers, we grouped them into one of three categories: beginner,
intermediate, and advanced user.

We then had the subjects complete a series of tasks. One evaluator explained the task to each of the subjects and answered any questions; the other evaluator recorded what the subject was doing, any comments that the tester made, and the time that it took the subject to complete the task. While the subject performed the tasks, we asked them to describe what they were doing and what they were thinking (speak aloud protocol). After the subject had completed the tasks, we asked them to answer a few questions regarding the overall usability of the system. Finally, we asked the subject to provide us with any suggestions that could be made to improve the tool.

The study uncovered that the GUI needed to be more intuitive and descriptive. The GUI was non-intuitive in the sense that when users tried to accomplish certain tasks, the interface did not respond in the way that they expected. An example of such a task is the creation of a flow line between two nodes: MonarchGraph expects the user to hold down the shift key and then to drag the mouse in order to create a link between nodes. Unfortunately, this behavior—and similar other non-intuitive GUI features—are predefined in MonarchGraph. Interestingly, for the computer-literate subjects, intuitiveness meant that microSynergy should behave much like GUIs that were already in use by them. If these subjects encountered non-intuitive GUI behavior, they found it difficult (and unnecessary) to think of alternative ways of performing the task.

3.2.2 Experiences

MonarchGraph follows an object-oriented application-framework approach. The framework provides hooks, which have to be implemented to provide application-specific functionality. The MonarchGraph distribution does not ship with the Java source code. A Javadoc API guide is provided that explains the interfaces and framework hooks.

To implement the framework hooks and the internal data model (abstract syntax graph for our DSVL), a fair amount of code has to be written (about 2500 LOC for the basic editor prototype and about 7500 LOC for the full editor). Thus, MonarchGraph’s approach is less well suited for rapid prototyping compared to DiaGen. Indeed, it took us three times as long to obtain a first basic editor prototype. However, further enhancements of the prototype were easily accomplished without hitting unexpected roadblocks and the maintenance of the new architecture was significantly easier.

Even though we had a working prototype with all of the planned functionality, the usability study showed us that significant changes in the GUI’s behavior would be necessary to get the tool adopted in an industrial setting. Unfortunately, MonarchGraph’s GUI behavior cannot be easily customized. For example, we could not change the standard behavior for link creation, which caused the aforementioned usability problems. Also, domain-specific error messages are hard to realize. Furthermore, as stated above, microSynergy should be adaptable to meet the requirements of many different users in different settings. Therefore, tool customization is a major concern. MonarchGraph, however, offers few mechanisms to make the GUI behavior customizable.

We concluded that the decision to adopt an "off-the-shelf" GUI component was a good decision in terms of software engineering considerations but the idiosyncrasies of the MonarchGraph GUI could (and probably would) put industrial adoption of our DSVL in jeopardy.
3.3 ACTD with Microsoft Visio

For the ACTD approach, we analyzed office tools with regard to their suitability in providing a development environment for a microSynergy interface with the specified usability features. We just started our implementation with Visio and report on preliminary results. Figure 4 shows a screenshot.

We decided to choose Microsoft Visio 2002 because it is highly customizable and offers a robust user interface to build upon. Furthermore, it has a large user base and is commonly found in industry. Visio can be easily customized for different domains as nicely illustrated by the applications that Visio already offers: Web maps to visualize the components of Web sites, ER diagrams to model databases, electrical engineering diagrams for industrial control systems etc.

![Figure 4: Screenshot of microSynergy in Visio](image)

A customized application for a specific domain can, for example, offer (1) customized stencils that contain the visual elements of the domain, (2) additional toolbars, accelerators and menus, (3) additional menu entries in a visual element’s context menu, (4) custom properties for visual elements, (5) windows that contain hierarchical views (“model explorer”) and domain-specific error messages, and (6) help features as an extension to the Visio help systems.

Visio exposes a VisualBasic API to access and analyze a document. All GUI elements (e.g., window, page, shape, and selection objects) are represented in the Visio object model. Thus, it should be possible to seamlessly integrate domain-specific functionality on top of Visio.
From the ACTD perspective, Visio promotes tool adoption by leveraging:

**a familiar GUI:** The user interacts with a familiar environment and paradigm. Application knowledge has been typically built up by the user over years. Since the user is already familiar with the standard functionality, he can concentrate on learning new functionality (incrementally).

**tool interoperability:** Other office tools such as Word, PowerPoint, and Excel interoperate with Visio via cut-and-paste and (file-based) import/export facilities. For example, a SDL drawing in Visio can be easily imported in Word for documentation and PowerPoint for presentation purposes. Thus, users can be more productive in their daily work. Visio can output drawings, templates, and stencils in XML encoding. Thus, standard XML tools can be used to pre- and post-process a document.

**customization and personalization:** Office applications often have fine-grained customization features, especially for the graphical user interface. For example, GUI elements in Visio can be (interactively) repositioned and hidden.

**tool support:** Popular tools come with a large infrastructure that provides useful information to the user. For example, (online) publications discuss how to use a tool most effectively. Mailing lists and discussion forums help troubleshoot users’ problems.

Tools that have a small target audience—which is typically the case for research tools—especially lack in the above areas. Since these tools focus often on the “proof of concept” and have limited financial resources and manpower, their GUIs provide only rudimentary functionality and does not support sophisticated customization or scripting. Since the user base is small, few experienced users exists (sometimes these users are mostly the tool’s developers) that can offer help via mailing lists or newsgroups. Documentation is unprofessional, outdated, or non-existent.

### 3.3.1 Experiences

Visio keeps a group of graphical elements in a stencil (see “Shapes” window on the left in Figure 4). For SDL, we could reuse several SDL shapes (masters in Visio terminology) that were already part of Visio. We had to modify Visio’s SDL “Output” master (see “Heat off” and “Heat on” in Figure 4) because its orientation was wrong. We used the master editor to change the shape and the master icon editor to change the shapes appearance on the stencil. We reused the SDL “Process” master (see “Heater” in Figure 4) from another SDL editor [SDL].

We found that domain-specific (GUI) behavior can be conveniently implemented with VisualBasic scripting. It was helpful having had prior experience in scripting other Microsoft office tools; they share many concepts and code can often ported with no or minor modifications. Since a programmer can incrementally add and immediately afterwards test the code, it is natural to build the application with rapid prototyping. VisualBasic offers a full development environment (with editor and debugger) that is tightly integrated with Visio. Scripting languages are a common mechanism to provide extensibility in office tools. Thus, power-users can look “under the hood” of a DSVL implementation and modify it to better fit their needs.

An important customization feature in Visio are custom properties. They are essentially key/value pairs that can be used to associate data (e.g., strings, numbers, boolean values, and lists) with visual elements. Custom properties are automatically saved and loaded as part of a Visio document. Thus, in our case, no code for persistence had to be written and the user can use the standard save and load functionality.
Shrink-wrapped applications such as Visio can provide strong support for editor functionality, but offer quite limited support otherwise. Esser and Janneck observe: “One key impediment to the use of visual languages in very domain-specific contexts is the high initial cost of creating a useful baseline environment (editor, possibly compiler/interpreter and debugger etc.)” [EJ01]. They propose a DSVL generator tool, Moses, to ease development and lower the cost of the baseline environment. ACTD’s approach is different. Suitable applications are selected with the goal of maximizing support for the baseline environment. Missing parts of the baseline environment and domain-specific functionality are implemented on top, typically with scripting.

4 Summary of Experiences

Figure 5 depicts a table that summarizes our implementation experiences. It states features that we deem important for DSVL development along with a rough evaluation whether the approach supports the feature well (+), average (⊗), or poorly (−). Note that the table is only indicative and represents our (limited) experiences in implementing a single DSVL.

<table>
<thead>
<tr>
<th>Feature</th>
<th>DiaGen</th>
<th>MonarchGraph</th>
<th>Visio</th>
</tr>
</thead>
<tbody>
<tr>
<td>user familiarity</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>user customization</td>
<td>−</td>
<td>−</td>
<td>+</td>
</tr>
<tr>
<td>rapid prototyping</td>
<td>+</td>
<td>⊗</td>
<td>+</td>
</tr>
<tr>
<td>standard editor functionality</td>
<td>⊗</td>
<td>⊗</td>
<td>+</td>
</tr>
<tr>
<td>flexibility</td>
<td>+</td>
<td>⊗</td>
<td>⊗</td>
</tr>
</tbody>
</table>

Figure 5: Summary of the Implementation Approaches

In contrast to both DiaGen and MonarchGraph, Visio offers a familiar user interface and is highly customizable. All approaches are suitable for rapid prototyping. However, more development effort is necessary in MonarchGraph to obtain a first, working prototype. Visio seems best suited for rapid prototyping because development starts with a complete (generic) graphical editor. Domain-specific functionality can then be incrementally implemented on top. Visio is a powerful graph editor with many features. In contrast, DiaGen and MonarchGraph have only rudimentary editor functionality. Since source code is available for DiaGen, it offers the greatest flexibility for the developer. However, this freedom is not for free; development and maintenance effort can become a major headache. Both MonarchGraph and Visio are restricted by the API that they offer. This does not pose a problem as long as the API allows the programmer to realize the desired functionality; however, if the API is not general or flexible enough, “workarounds” have to be found or one has to give up some functionality that was originally planned. Worse, these problems are hard to predict during the design and then surface later at the development phase.

We argued that idiosyncratic GUIs along with unfamiliar concepts and paradigms are the primary adoption challenges that have to be overcome for DSVLs. Our main hypothesis is that users will more likely use DSVLs that are integrated into an environment that they use daily and know intimately. ACTD’s approach to achieve this goal is building DSVLs as extensions of shrink-wrapped office tools.

We used the microSynergy DSVL as a case study to discussed this approach and contrast it with two other, more common implementation approaches. As opposed to the non-ACTD
implementations, the ACTD implementation with Visio has many desirable features from the user and adoption point of view.

This study gives a first indication that ACTD can indeed help tool adoption. However, formal studies with user experiments are needed. A challenging problem for such a study is, for example, how to measure adoption. We also hope that other DSVL researchers will adopt ACTD and report on their experiences.

Acknowledgments

We thank Yury Bychkov, Marc d’Entremont and Andrew McNair for their support in developing microSynergy. Thanks to Crina-Alexandra Vasiliu for proofreading an earlier version of the paper. Finally, thanks to the anonymous reviewers for valuable feedback.

The microSynergy project has been jointly funded by Intec Automation Inc. and the Advanced Systems Institute of British Columbia.

References


