Extending Programming Environments to Support Architectural Design

Spiros Mancoridis and Richard C. Holt

Department of Computer Science
University of Toronto
Toronto, Ontario M5S 1A4, Canada

Abstract

As software systems grow in size and complexity, the demand for languages and tools to capture higher-order abstractions than those supported by programming languages increases. One of these abstractions is the architectural design, which specifies a system's components, their interfaces, and their interrelationships using textual or visual notations. Although there have been significant advances in programming languages and environments, research into languages and tools for architectural design is still preliminary. Moreover, there has been little emphasis on integrating design tools with existing programming environments. This paper describes how the Object-Oriented Turing programming environment was extended to accommodate languages and tools for specifying and visualizing architectural designs.

1 Introduction

Programming environments are tightly integrated tool sets that support the production of source code in a programming language. They usually comprise tools for editing (often syntax directed), compiling, executing, debugging, and source code browsing. These environments, however, do not necessarily support notations and tools for specifying and viewing abstractions of higher-order than those supported by programming languages. Common programming language abstractions are data types, procedures, modules, and classes. When a system expands to include many instances of these abstractions, it becomes desirable to aggregate them into higher-level ones. These higher-level abstractions help developers cope with the size and complexity of software systems, which is important both during development and future maintenance.

Examples of programming environments[1] include Smalltalk [11] and Interlisp [28] from Xerox Parc, Integral C [25] from Tektronics, and the Turbo environments from Borland such as that for Prolog [1]. A well known programming environment developed at Brown University, called Pecan [24], is credited with the innovation of multiple views. Pecan supports diagrammatic views for expression trees, data types, symbol tables, and so on, but does not support views for higher-level abstractions.

Although significant research in programming languages and their respective environments has been conducted, research in languages and tools for architectural design is still preliminary. The need for architectural design languages was first expressed by DeRemer and Kron [7]. They coined the terms Programming-in-the-Small (PitS), the activity of producing source code in a programming language, and Programming-in-the-Large (PitL), the activity of specifying the interconnections between PitS entities (*i.e.,* types, variables, procedures). The product of PitL is the architectural design. Our definition of architectural design is close to the definition of software architecture[2] given by Schwanke et al. [26], who view architectures as the permitted or allowed set of connections among components.

Module Interconnection Languages (MILs), such as those defined by DeRemer and Kron [7], Cooprider [5], and Tichy [29], represent early attempts to define languages for specifying architectural designs. MILs are layered on top of common programming languages. Their advantage is that they closely couple the design specification to the source code, making the specification amenable to mechanical processing (specifications can be compiled). More recent MILs by Penny [21] and Schwanke et al. [26] have a graphical rather than textual syntax, making them more appealing to software designers who are accustomed to using diagrams. These visual MILs are related to the visual notations found in CASE tools, which evolved separately as a set of diagrammatic conventions to support various design approaches. Examples of such notations are Jackson's System Design [15] and Booch's Object-Oriented Design [3]. These CASE notations differ from the visual MILs in that they are often informal, closely tied to particular software development process, and detached from the implementation programming language.

[1] For surveys of programming environments, and software engineering environments in general, readers are directed to papers by Dart et al. [6], Perry and Kaiser [22], and Mancoridis [17], as well as a book titled Software Engineering Environments [4] by Brown, Earl, and McDermid.

[2] The term *software architecture* [23, 10] is more widely used in a broader sense than architectural design.

We are interested in languages and tools to support both programming and architectural design. Since these activities are related, we believe that extending well-established programming environments to accommodate tools for architectural design is important and interesting. Section 2 focuses on the programming activity by describing a programming language, called Object-Oriented Turing (OOT), and its respective environment. Section 3 focuses on the architectural design specification activity by presenting an overview of languages for specifying and visualizing architectural designs. The tools supporting these languages are then presented in Section 4. Section 5 describes how a translator, called Star, was used to extend the OOT programming environment to support tools for architectural design specification and visualization. Section 6 outlines the architectural design of the Star translator. Finally, Section 7 presents some closing remarks and identifies future research opportunities.

2 The OOT Programming Language and Environment

OOT is the object-oriented extension of the Turing programming language [13]. It is a high-level programming language that is well suited for both teaching programming as well as developing industrial-strength software. OOT has all of the features of Pascal as well as modules, string handling facilities, type-safe variant records, and dynamic arrays. OOT also has features supporting concurrency and object-oriented programming. For concurrency, developers can use language supported monitors. For object-oriented programming, OOT provides classes, inheritance, and polymorphism. Finally, OOT can be used as an alternative to C for systems programming. The language has all of the features of C while encouraging a safe and reliable programming style.

The OOT language is supported by a programming environment[3] [19] that comprises a tightly-integrated set of tools for editing, high-speed compiling, linking, executing, and debugging OOT programs, as well as for browsing the file system. An important aspect of the OOT environment is its consistent user-interface. To the user, OOT consists of a number of windows easily identifiable by variations in their colour, size, screen position, and title. OOT's support of a rapid edit-compile-link-debug cycle is similar to that of Borland's Turbo environments.

Figure 1 shows a snapshot of the OOT environment in execution. The top-left window is the Control Panel, to its right is the Directory Viewer for browsing through the file system and opening files. Below the Control Panel is the File Viewer for editing text. The two windows beneath the Directory Viewer are run-time windows for displaying the output of the executing OOT program, which in this case is the *MiniTunis* operating system.

Finally, the window on the bottom left of Figure 1 is the Interface Viewer, which is used for displaying the signatures of all exported entities (*e.g.*, functions, procedures, types, variables, constants) of OOT units (*i.e.*, module, class, monitor) along with textual annotations describing the signatures. Clicking on the name of an exported entity in the Interface Viewer displays the signature of that entity together with its associated textual annotations. Double clicking on one of these names causes the source code of that signature to pop up in an OOT File Viewer. The information displayed in the Interface Viewer is stored as a separate IFC file (one interface file per OOT unit) that is created automatically by the Star translator (described later in Section 5) from the OOT source files.

Having presented an overview of the programming language and its environment, we next describe the languages for specifying and visualizing architectural designs.

3 Architectural Design Languages

This section presents brief descriptions and examples of languages for specifying and visualizing architectural designs, called SIL and DL respectively.

3.1 The Subsystem Interconnection Language (SIL)

SIL is a textual language for specifying architectural designs. The basic unit of SIL is the *subsystem*, which is an abstraction used for aggregating collections of related components in a software system. SIL has a syntactic construct for composing hierarchies of subsystems from other subsystems and OOT units; it also has constructs for specifying dependencies between subsystems, via the *import* clause, and hiding information, via the *export* clause. The SIL semantics are given by a set of design rules (written in first-order logic and Prolog) to ensure the correct use of SIL constructs. Examples of such rules include that: subsystems must only export entities that they contain, subsystems must not contain themselves, and so on. A detailed formal definition of the syntax and semantics of SIL has been described elsewhere [14].

Most architectural design languages have mechanisms for composing systems made up of nested modules; the only relationships between these modules are the *provides-requires* dependencies between their PitS entities. SIL relations are not based such dependencies, but rather on relationships between coarsely-grained PitL elements, such as entire modules and subsystems. Another difference is that SIL is not restricted to a fixed number of relations. SIL currently supports relations such as *contain*, *import*, and *export*, and may later be augmented with other relations to support inheritance, configuration management, dataflow, and so on.

3.2 The Drawing Language (DL)

Harel [12] predicts that more future languages will be diagrammatic and encourages the use of visualizations when dealing with complex systems. We agree that visual representations of architectural designs are

[3]The OOT programming environment is currently implemented on Unix platforms such as `Sun/4`, `RS/6000` and `SGI`, as well as on `MS-DOS` and `MS-Windows` platforms for personal computers. For details on how to get a demo of OOT, interested readers may obtain the *ootDistrib* file from the `pub` directory of our `ftp` site `128.100.1.192` on the Internet.

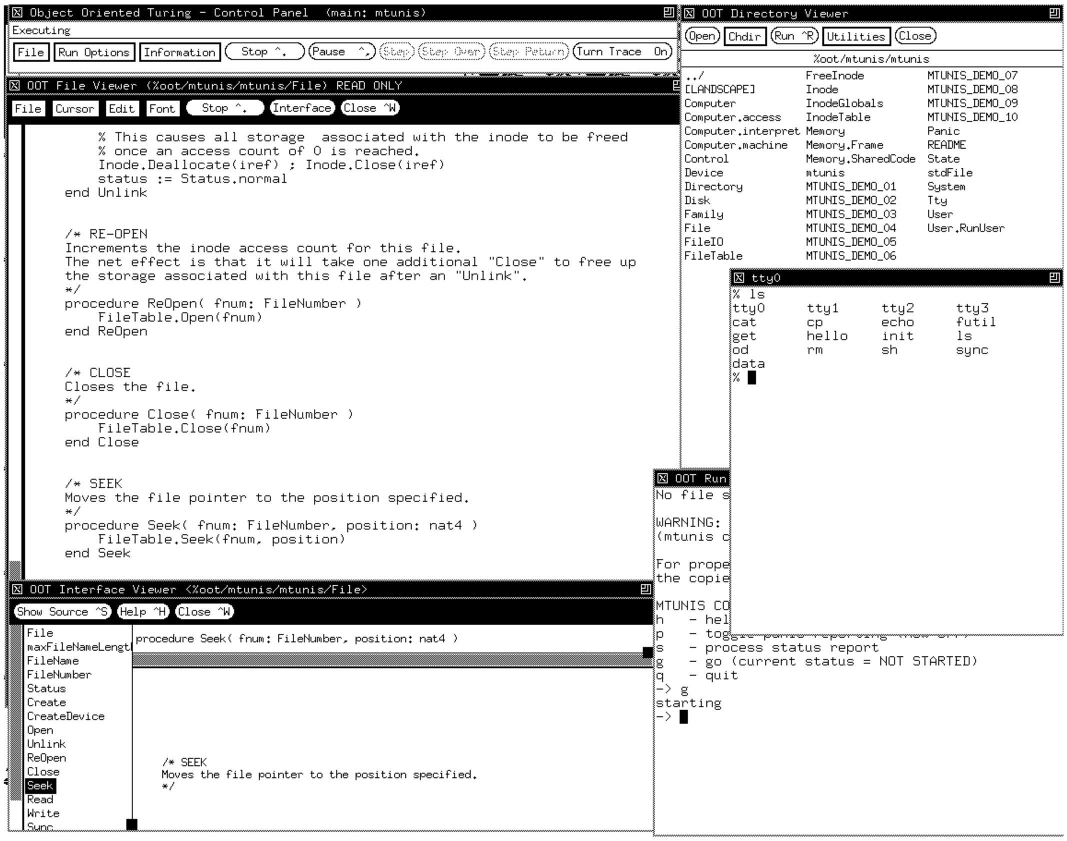

Figure 1: **A Snapshot of the OOT Programming Environment**

often more intuitive and easier to understand than their textual counterparts. We designed the DL language for specifying drawings made up of annotated coloured boxes and arrows. Apart from syntactic constructs for specifying nested boxes, arrows, colours, and textual annotations, DL has constructs for specifying linear transformations (translations and scalings) on geometric objects to facilitate panning and zooming. DL specifications also have semantic constraints that disallow diagrams from having boxes with overlapping boundaries, disallow arrows that are not attached to boxes, and so on. A detailed formal definition of the syntax and semantics of DL has been described elsewhere [14]. Examples of similar drawing languages include the Graph Exchange Format (GXF) [8] for visualizing directed labelled graphs, and the Graph Description Language (GDL) [30] for visualizing graphs in three dimensions.

We use DL specifications to describe visualizations of architectural designs written in SIL. Having two separate languages is not strictly necessary since SIL can be augmented with clauses for specifying layout and graphical information. However, factoring the two languages creates a modular framework that enables us to experiment with different architectural design languages independently of the various visual notations used to represent designs.

3.3 Examples

So far we have described languages for programming (OOT) as well as for specifying (SIL) and visualizing (DL) architectural designs. Figure 2 shows examples written in each of these languages as they are used to describe the architectural design depicted visually at the bottom-left corner of the figure. The white boxes represent *subsystems*, dark[4] boxes represent OOT units, dark arrows represent *import* relations between subsystems, light arrows represent dependencies between OOT units, nested boxes represent *contain* relations, and light frames around boxes represent *export* relations. This architectural design consists of two subsystems *S1* and *S2*. The OOT module *B* in *S1* uses module *A*, also in *S1*. Module *A* uses module *C* in subsystem *S2*. For this to be allowed, the parent subsystem of *A (S1)* must import the exported module *C*. This is a design constraint imposed by the semantics of the SIL language described in detail elsewhere [14].

In Figure 2, the top-left box is the OOT specification with the actual source code that is compiled and executed. Next to it is the SIL specification, which consists of subsystems and their interrelationships. Finally, the top-right box is the DL specification, which comprises graphical and layout information.

In the next section, we present tools that support the architectural design languages.

4 Architectural Design Tools

This section describes two tools; the first tool is a visual editor for specifying diagrammatic representations of architectural designs, the second tool is a

[4] In a colour diagram the dark boxes would have been blue, light arrows blue, and light frames green.

Object-Oriented Turing Programming Language (OOT)

```
module A
   import C

   % source statements
end A


module B
   import A

   % source statements
end B


module C
   % source staments
end C
```

Subsystem Interconnection Language (SIL)

```
subsystem S1
   contain A, B
   import C
end S1


subsystem S2
   contain C
   export C
end S2
```

Drawing Language (DL)

```
box S1
   xorigin 0 yorigin 0 scale 1
   boxcolor white titlecolor black

   include A
     x 10 y 20 width 5 height 5
     framecolor black
   end A

   include B
      x 20 y 10 width 5 height 5
      framecolor black
   end B

   edge
      edgecolor black  target C
      exit side right position 0.5
      entry side bottom position 0.5
end S1

box S2
   xorigin 0   yorigin 0   scale 1
   boxcolor white  titlecolor black

   include C
      x 20 y 10   width 5 height 5
      framecolor green
   end C
end S2

box A
   xorigin 0 yorigin 0 scale 1
   boxcolor blue titlecolor white

   edge
      edgecolor blue  target C
      exit side right position 0.5
      entry side left position 0.5
endA

box B
   xorigin 0 yorigin 0 scale 1
   boxcolor blue titlecolor white

   edge
      edgecolor blue target A
      exit side top position 0.5
      entry side bottom position 0.5
end B

box C
   xorigin 0 yorigin 0 scale 1
   boxcolor blue titlecolor white
end C
```

Visual Representation of the DL Specification

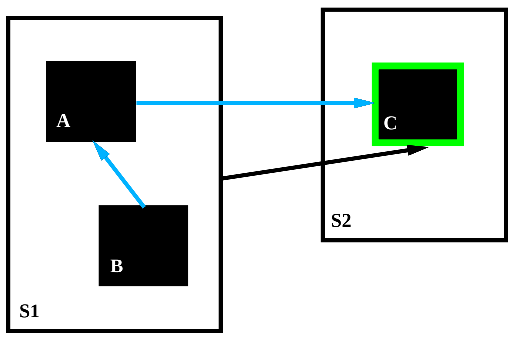

Figure 2: **Examples of the OOT, SIL, and DL Languages**

Prolog interpreter for automatically checking architectural designs for semantic well-formedness.

4.1 Visual Editor

An architectural design written in SIL can be created using the editor in the OOT File Viewer, which was mentioned in Section 2. Its corresponding visual representation can be created using the visual editor. The visual editor stores each diagram as a DL text file which maintains the names, shapes, colours, and positions of all entities in the design diagram. The visual editor supports the drawing and mouse-based manipulation of annotated multi-coloured boxes and arrows. It also supports panning and zooming functions that facilitate the navigation through large design diagrams. Developers can zoom in and out of diagrams by opening a new edit window with a subset of a diagram, by diving into or out of a box by clicking on it, or by gradually enlarging or reducing a diagram using mouse-based zoom-in and zoom-out. As a mechanism for handling complexity, the visual editor also supports the elision of arrows and contents of subsystems.

Figure 3 is a snapshot of the visual editor depicting the architectural design of the Star translator (described later in Section 5). In Figure 3, dark boxes represent atomic OOT units, white boxes represent subsystems, and grey boxes represent subsystems whose contents have been elided. Boxes with dotted frames are exported from their parent subsystems. The arrows representing subsystem import dependencies have been elided. The arrows shown in Figure 3 are dependencies between atomic OOT units. When elided subsystems contain atomic units participating in dependencies, the arrows point to or from their elided (grey) subsystem ancestor boxes instead of the intended atomic unit boxes. For example, the OOT unit ootParser in subsystem Parsers depends on an OOT unit in the elided subsystem GLOBALS.

The visual editor does not ensure that a diagram satisfies the semantic constraints of DL, just as a general purpose text editor does not check a program for errors. The next section describes how checking whether a diagrams satisfies the semantic constraints of DL can be done using a Prolog interpreter.

4.2 Prolog Interpreter

An architectural design can be expressed as a set of Prolog facts. The following section describes how SIL and DL specifications can be automatically translated into sets of such facts. For example, if a subsystem S contains a module M, the following facts should be part of the complete Prolog specification:

```
subsystem('S').
module('M').
contain('S','M').
```

The semantic constraints for both SIL and DL are specified using sets of Prolog rules, one set for each of the two languages. For example, the constraint stating that two entities are visible to each other if there is a parent entity that directly contains both of them, is described with the following Prolog rule:

```
seeSibling(A,B) :- contain(P,A),
                   contain(P,B).
```

Along with the constraint rules are the deductive rules for computing facts such as the transitive closure, *containPlus*, of the *contain* relation:

```
containPlus(X,Y) :- contain(X,Y).
containPlus(X,Y) :- contain(X,Z),
                    containPlus(Z,Y).
```

We use a Prolog interpreter as a theorem prover for verifying that instances of SIL and DL specifications, translated into Prolog facts, do not violate the semantic constraints given by the Prolog rules. In another paper [18] we showed how other interpreters, such as the one in the ConceptBase [16] Knowledge Representation Management System, can be used instead of Prolog.

This concludes the description of the tools (visual editor and Prolog interpreter) that support architectural design. We next present a translator, called Star, which coordinates the data of these tools to achieve their integration with the OOT programming environment.

5 The Star Translator as a Tool Integration Mechanism

The tools supported by the OOT environment as well as the visual editor and Prolog interpreter, operate on their own private text files. An overview of the tools and their integration is shown in Figure 4. The text files are represented as ovals; the tools themselves are represented as rectangles. Using the text editor of the OOT environment, developers can manipulate both OOT source code and SIL specifications as text files. Each of these files may have a corresponding IFC file that is processed by the Interface Viewer. Visual representations of architectural designs, created using the visual editor, are stored as DL text files. Finally, the Prolog interpreter is used for verifying the consistency of SIL and DL specifications against the semantic rules of their respective languages.

In the center of Figure 4 is the Star translator, which coordinates the information that is distributed among the various file formats. The Star translator gathers the information from each language specification into a centralized "in-core" data structure that reflects the structure of the architectural design. Star can then translate specifications written in one language to a corresponding specification written in another language. The translations supported by Star are:

- *(SIL or OOT) to DL:* Star can generate a DL specification from OOT source code and SIL specifications. The Star translator has mappings that enables it to assign default colours and shapes to various OOT and SIL entities and relationships. It also executes the Sugiyama [27] automatic graph layout algorithm [2] for assigning positional information to each entity while minimizing the edge crossings between them. When

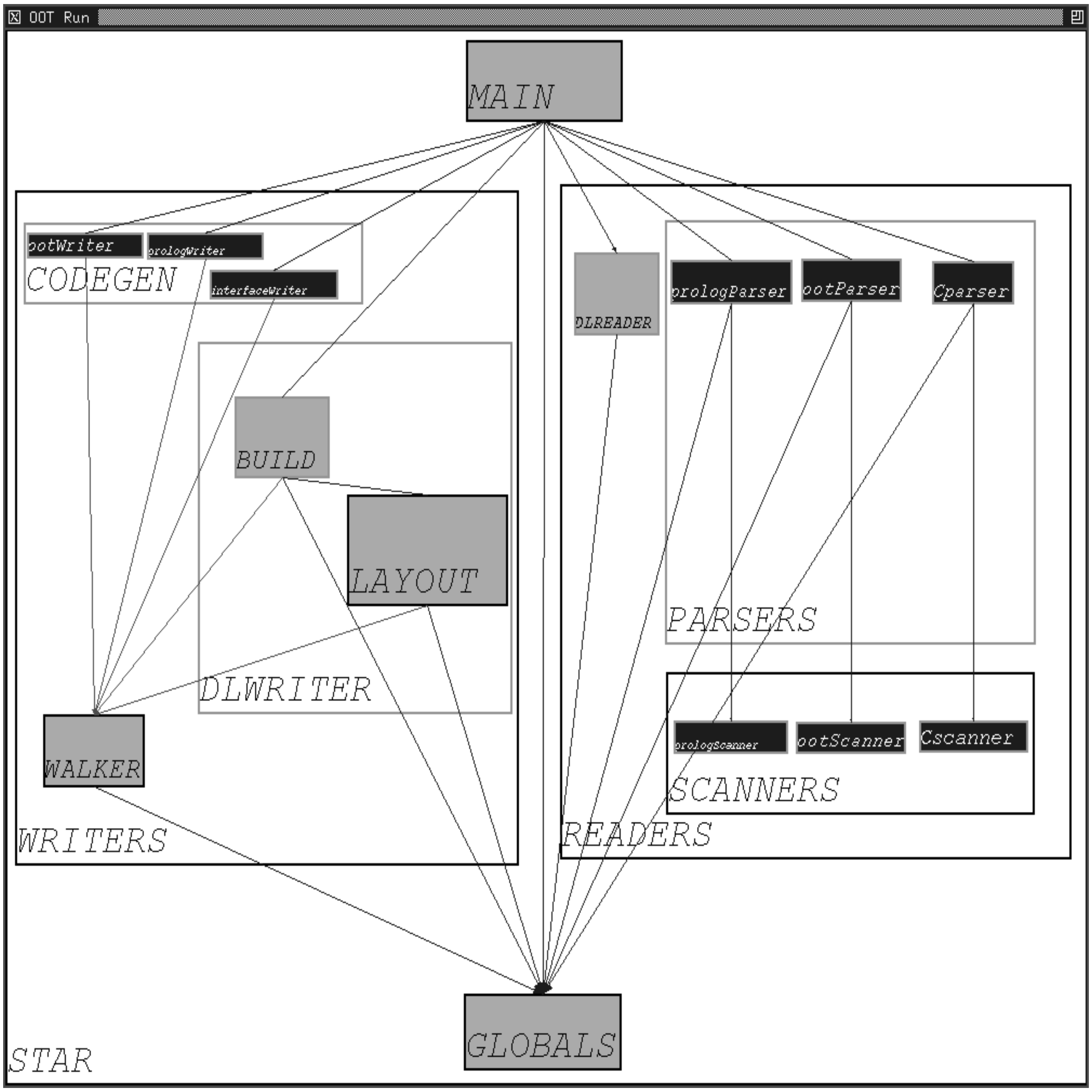

Figure 3: **The Visual Editor Displaying an Architectural Design**

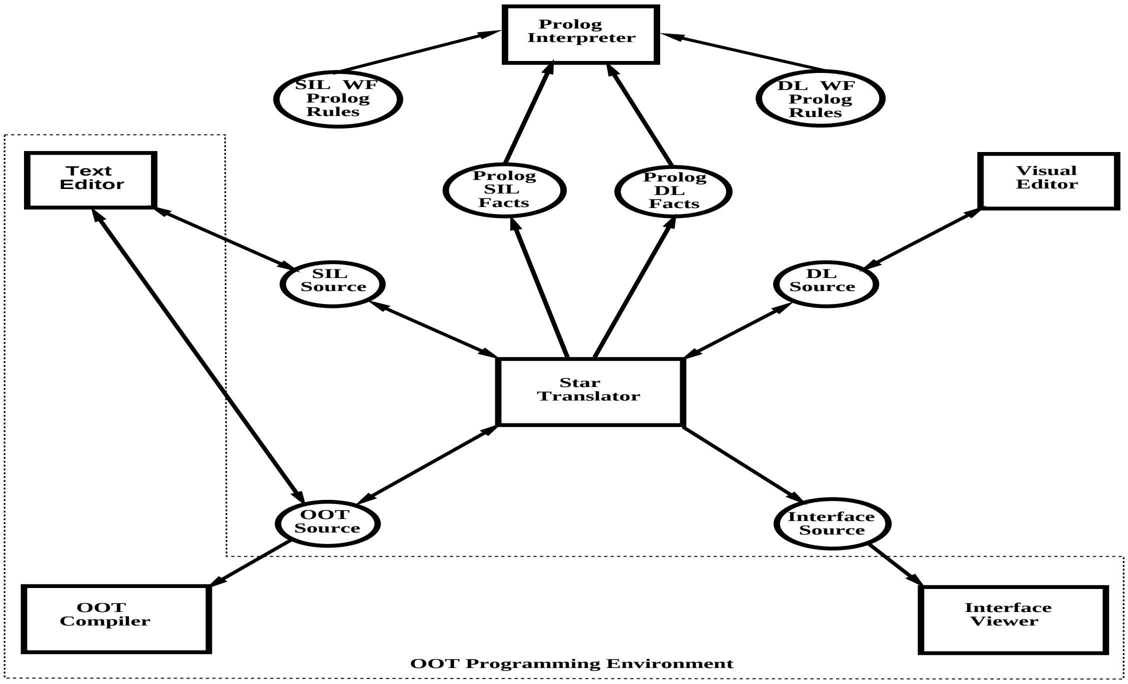

Figure 4: **Overview of the Tools and their Integration**

used in this bottom-up fashion (*i.e.*, architectural design visualizations are extracted from source code), the Star translator acts as a reverse engineering system to Müller's Rigi environment [20].

- *DL to (SIL or OOT):* Star can generate SIL and partial OOT specifications using information from DL specifications. The translation is partial because Star can infer structural information from a visual design but (obviously) it cannot infer the actual source code. When used in this top-down fashion (*i.e.*, code templates are generated from an architectural design diagram), the Star translator acts as a CASE environment with "code-generation"[5] capabilities similar to ObjectMaker [31].

- *(SIL or OOT) to IFC:* Star can parse SIL and OOT specifications to extract exported signatures, the location of the signatures in their respective file and any comments associated with them. Star then emits this information in a format that can be processed by the Interface Viewer.

- *(SIL or DL) to Prolog:* Star can parse SIL and DL specifications and extract structural information that would be of use to the constraint checker. Star can subsequently emit this information as Prolog facts that can be processed by the Prolog interpreter to determine the consistency of a specification against the SIL and DL rules.

The translations described above are achieved by parsing a particular persistent representation and creating the central Star data structure. Once created, this data structure is traversed and source code for another language is generated. This technique works because the tools share a common data schema which reflects the structural dependencies of OOT units and SIL subsystems.

Figure 5 illustrates the data integration among the tools through the common data schema. The box in the center represents the common schema populated with code (C), interfaces (I), and graphical (G) information. Each tool requires only a projection or view of this information. The SIL view needs the code of subsystems. The OOT view needs the code of OOT units. The DL view needs the layout and colour information. Finally, the Interface view needs information pertaining to exported signatures of OOT units and SIL subsystems. Note that not all information for each view is necessarily present in the data structure. For example, if the Star data structure is created by parsing a DL specification, there will be no information pertaining to OOT unit interfaces because such information is not present in the architectural design diagrams.

The next section describes an overview of the ar-

[5] The use of the term code generation in CASE tool advertisements is somewhat misleading. What is meant is that code templates can be generated automatically from the diagrams.

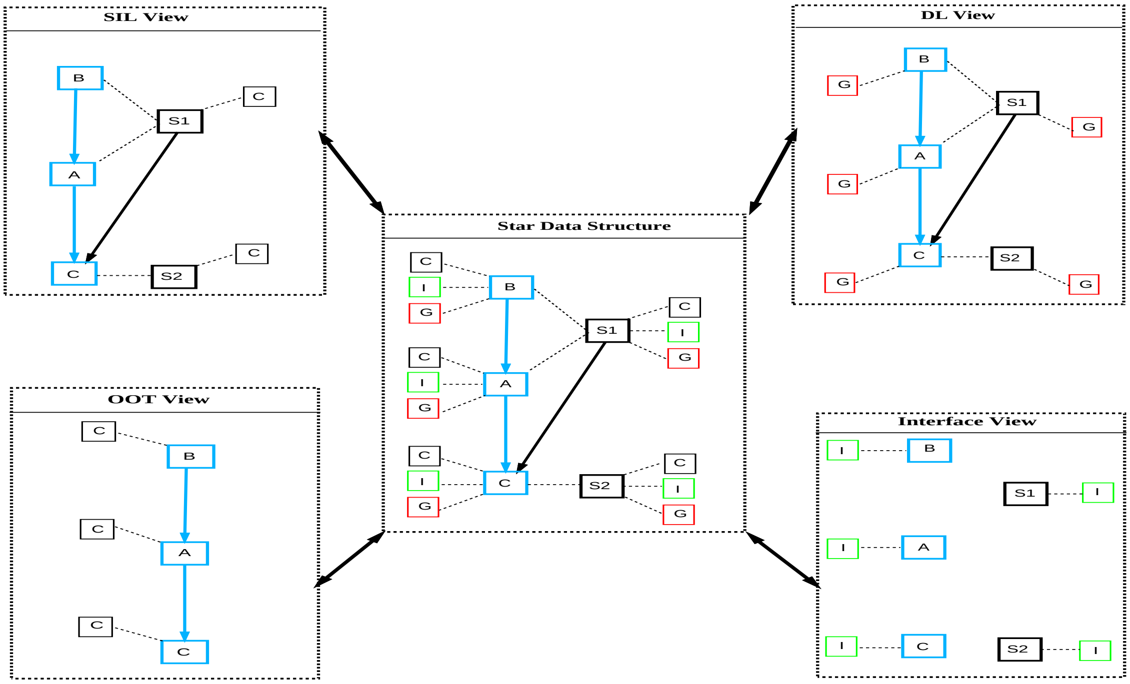

Figure 5: **Four Views of Star's Common Data Structure**

chitectural design of the Star translator.

6 The Architectural Design of the Star Translator

The Star translator is implemented in Object-Oriented Turing. A visual rendition of its architectural design is shown in Figure 3. The architectural design indicates that Star has two major subsystems, READERS and WRITERS The READERS subsystem contains subsystems for scanning and parsing specifications for the various languages. The WRITERS subsystem contains subsystems for traversing the Star data structure (WALKER) generating code (CODEGEN and BUILD), and laying-out DL diagrams (LAYOUT). The code in subsystem READERS translates a specification into the internal Star data structure, while the code in subsystem WRITERS traverses the Star data structure and emits code in other languages. The supporting subsystems, MAIN and GLOBALS, contain code for the user-interface and data structures of Star, respectively. The architecture of Star is modular, enabling a new tool to be added to the environment by implementing an appropriate parser and code generator for that tool. Current work to extend the functionality of Star to support C++ is almost complete.

This concludes the description of the architecture of Star. The next section summarizes the paper and discusses future research.

7 Conclusions and Future Research

Languages and tools for architectural design become increasingly important as software systems grow in size and complexity. In this paper, we presented languages and tools for specifying and visualizing architectural designs and showed how they can be integrated with programming languages and their respective environments.

Our approach to tool integration is based on translating the persistent data of each tool from one form to another via a centralized translator (Star). Another possibility would have been to integrate these tools more tightly via a data integration mechanism similar to the one provided by the PCTE repository [9]. This would imply defining a PCTE schema corresponding to the schema of the Star data structure, as well as adapting the tools to access the data in the repository.

Current research is underway to extend the design languages to support object-orientation, reuse libraries, configuration management, and data-flow. We are also looking into visualization issues for navigating and automatically laying-out large and complex graphs representing architectural designs.

Most developers would agree that modern programming languages and their respective integrated environments are helpful. Unfortunately, the same cannot be said about MILs and CASE tools which, despite their contribution in furthering our ability to specify software designs, have not gained the same degree of acceptance from developers as programming languages and environments have. Unlike textual MILs, which were short-lived[6], visual CASE tools are being used, albeit often in isolation from programming environments. This isolation makes it difficult to consistently share information between CASE tools and programming environments, which eventually results in outdated designs with respect to the implementation source code.

With this work we have tried to motivate the importance of specifying and visualizing high-level abstractions of software systems. Additionally, we wanted to demonstrate how to integrate tools that support these high-level abstractions with established programming environments.

References

[1] Bannister, H. Borland Introduces Turbo Prolog, Version 1.1. *InfoWorld 8*, 39 (September 1986).

[2] Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. Algorithms for Drawing Graphs: An Annotated Bibliography. Tech. rep., Department of Computer Science, Brown University (available via anonymous ftp from `wilma.brown.cs.edu 128.148.33.66` in file `/pub/papers/compgeo/gdbiblio.ps.Z`), November 1993.

[3] Booch, G. *Object–Oriented Design with Applications.* Benjamin/Cummings Publishing Company Inc., Redwood City, California, 1991.

[4] Brown, A. W., Earl, A. N., and McDermid, J. A. *Software Engineering Environments Automated Support for Software Engineering.* The McGraw–Hill International Series in Software Engineering, 1992.

[5] Cooprider, L. W. The Representation of Families of Software Systems. Tech. Rep. CMU–CS–79–116, Computer Science Department CMU, April 1979.

[6] Dart, S. A., Ellison, R. J., Feiler, P. H., and Habermann, A. N. Software Development Environments. *IEEE Computer* (November 1987), 18–28.

[7] DeRemer, F., and Kron, H. H. Programming–in–the–Large Versus Programming–in–the–Small. *IEEE Transactions on Software Engineering 2*, 2 (June 1976), 80–86.

[8] Eigler, F. C. GXF: A Graph Exchange Format. In *Declarative Database Visualization: Recent Papers from the Hy+/GraphLog Project*, A. Mendelzon, Ed. Technical report CSRI–285, University of Toronto, July 1993, pp. 91–107.

[6] Some features of MILs, such as nested modules, have made their way into modern programming languages.

[9] Gallo, F., Minot, R., and Thomas, I. The Object Management System of PCTE as a Software Engineering Database Management System. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Palo Alto, California, December 1986), pp. 12–15.

[10] Garlan, D., and Shaw, M. Architectures for Software Systems. In *tutorial given at ACM SIGSOFT '93: Symposium on the Foundations of Software Engineering* (Los Angeles, California, December 1993).

[11] Goldberg, A. *Smalltalk–80: The Interactive Programming Environment.* Addison–Wesley Publishing Company, Reading, Massachusetts, 1983.

[12] Harel, D. On Visual Formalisms. *Communications of the ACM 31*, 5 (May 1988), 514–530.

[13] Holt, R. C., and Cordy, J. R. The Turing Programming Language. *Communications of the ACM 31*, 12 (December 1988), 1410–1423.

[14] Holt, R. C., and Mancoridis, S. A Framework for Specifying and Visualizing Architectural Designs. Tech. Rep. CSRI–300, Computer Systems Research Institute, June 1994.

[15] Jackson, M. *Principles of Program Design.* Academic Press, New York, New York, 1975.

[16] Jarke, M. ConceptBase V3.0 User Manual. Tech. Rep. MIP–9106, Universitat Passau, March 1991.

[17] Mancoridis, S. A Multi-Dimensional Taxonomy of Software Development Environments. In *Proceedings of the 1993 IBM CASCON Conference* (October 1993), pp. 581–591.

[18] Mancoridis, S., Holt, R. C., and Godfrey, M. W. A Program Understanding Environment Based on the "Star" Approach to Tool Integration. In *Proceedings of the Twenty–Second ACM Computer Science Conference* (March 1994), pp. 60–65.

[19] Mancoridis, S., Holt, R. C., and Penny, D. A. A "Curriculum–Cycle" Environment for Teaching Programming. In *Proceedings of the Twenty–Fourth ACM SIGCSE Technical Symposium on Computer Science Education* (February 1993), pp. 15–19.

[20] Müller, H. A. Rigi as a Reverse Engineering Tool. Tech. Rep. DCS–160–IR, University of Victoria, March 1991.

[21] Penny, D. A. *The Software Landscape: A Visual Formalism for Programming–in–the–Large.* PhD thesis, Department of Computer Science, University of Toronto, 1992.

[22] PERRY, D. E., AND KAISER, G. E. Models of Software Development Environments. In *Proceedings of the 10th IEEE International Conference on Software Engineering* (Singapore, 1988), pp. 60–68.

[23] PERRY, D. E., AND WOLF, A. L. Foundations for the Study of Software Architectures. *Software Engineering Notes 17*, 4 (October 1992), 40–49.

[24] REISS, S. P. PECAN: Program Development Systems that Support Multiple Views. *IEEE Transactions on Software Engineering 11*, 3 (August 1985), 276–285.

[25] ROSS, G. Integral C—A Practical Environment for C Programming. In *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (December 1986), pp. 42–48.

[26] SCHWANKE, R. W., ALTUCHER, R. Z., AND PLATOFF, M. A. Discovering, Visualizing, and Controlling Software Structure. In *Proceedings of the Fifth International Workshop on Software Specification and Design* (Pittsburgh, Pennsylvania, May 1989), pp. 147–150.

[27] SUGIYAMA, K., TAGAWA, S., AND TODA, M. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics 11*, 2 (February 1981), 109–125.

[28] TEITELMAN, W., AND MASINTER, L. The Interlisp Programming Environment. *IEEE Computer 14*, 4 (April 1981), 25–33.

[29] TICHY, W. F. Software Development Control Based on System Structure Description. Tech. Rep. CMU–CS–80–120, Computer Science Department CMU, January 1980.

[30] WARE, C., HUI, D., AND FRANCK, G. Visualizing Object Oriented Software in Three Dimensions. In *Proceedings of the 1993 IBM CASCON Conference* (October 1993), pp. 612–620.

[31] WILLIAMS, T. Object–Oriented CASE Tool Lets User Tailor His Own Methods. *Computer Design* (September 1991), 122.