# Loosely Integrating Tools Using the Star System

Spiros Mancoridis

Department of Computer Science
University of Toronto
Toronto, Canada

## Abstract

A developer must often be able to understand a software system at several levels of detail — from abstract system architecture down to source code — before being able to change system functionality or fix bugs. Each of these levels might require its own notation and tools. For example, source code, written in a programming language, is viewed and manipulated using browsers and editors, whereas architectural designs, expressed as diagrams, are viewed and manipulated using visual editors.

In this paper we show how a multi-view perspective of a software system can be realized, simply and economically, by loosely integrating a set of existing independent tools. Ideally, all environments would be designed *a priori* to allow their combination into tightly integrated systems. In actual fact, it is common that developers of individual tools cannot foresee the ways in which their tools will be used. People who integrate such tools must exploit what is available to them. The Star system demonstrates how loose integration among existing tools can rapidly and simply result in useful environments for software development. [1]

## 1 Introduction

A developer must often be able to understand a software system at several levels of detail. In this paper we describe how we can provide software developers with a loosely integrated *Software Development Environment (SDE)* that supports multiple views of software by integrating several existing tools.

This SDE was engineered economically by the loose integration of several existing tools, some of which were originally developed to support activities unrelated to software development. The integration mechanism is based on the Star approach, whereby tools communicate by translating their persistent data into representations processable by other tools.

We should distinguish the Star approach from other approaches such as environment frameworks described in a book by Brown et al. [2]. These frameworks, although desirable, are large, complex systems that can be time consuming to obtain, access, and maintain. The Star system provides a simple mechanism for integrating existing tools to produce SDEs that can be used immediately.

Our presentation of the material is "bottom-up". We start by describing each individual tool in Section 2. In Section 3 we show how these tools were integrated using Star. In Section 4 we argue that the individual tools, when used in concert in an integrated environment, support software development. Finally, in Section 5, we make some conclusions and address future research opportunities.

## 2 The Independent Tools

In this section we present a concise overview of the individual tools that comprise our SDE. Each of these tools is really an environment in itself; however, we will not refer to them as such to avoid any confusion with the SDE, the product of their integration. There are three such

[1] The author is supported by a fellowship from IBM Canada. The IBM contact for this paper is Jacob Slonim, Centre for Advanced Studies, IBM Canada Ltd.

tools: **OOT**, a programming system; **ConceptBase**, a knowledge base management system; and **Hy$^+$**, a data visualization system.

## 2.1 OOT: A Programming System

Object-Oriented Turing (OOT) is a programming system [11] for the object-oriented extension of the Turing language [7]. It comprises a tightly integrated set of tools for editing, compiling, executing, and debugging OOT programs, as well as for browsing the Unix file system. An important strength of the OOT system 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.

To support large-scale software development, the OOT system was extended by two additional tools that provide higher-level, abstract views of OOT programs. The first tool is called the *Interface* viewer and the second the *Software Landscape* viewer [13]. These new tools, unlike the other ones mentioned, are loosely integrated with the OOT system through shared files. Figure 1 shows a snapshot of the OOT system in execution.

### 2.1.1 Interface Viewer

The OOT Interface viewer tool is used to display the signatures of all exported entities (*e.g.*, functions, procedures, types, variables, constants) of an OOT unit (*i.e.*, module, class, monitor, or subsystem) along with textual annotations describing them. This information is stored persistently as a separate file (one interface file per OOT unit) and is interpreted by the viewer tool.

The interface files are created manually by developers from the source code and stored in files. The interface information serves as a form of documentation and can be used for future system maintenance.

### 2.1.2 Software Landscape Viewer

The Software Landscape viewer is a graphical editor that operates on *Software Landscapes*; it allows developers to document the architecture of OOT systems by creating and manipulating Landscapes.

The *Software Landscape* [8, 10] is a notation for relating graphical software entities, such as designs, with non-graphical entities, such as source code, in a controlled, integrated, and consistent manner. A distinguishing feature of the Software Landscape is its formally defined rules of well-formedness [13].

The term "Landscape" is intended to connote the idea of a single visual framework within which all software development occurs; the products of software development remain in the Landscape to be explored, understood, and used by others. These products are stored as variously related entities. This stored information may be viewed and manipulated visually and in colour on a large display screen.

The Landscape is meant to aid large-scale programming. The lowest-level entities are programming language constructs such as modules and classes. Higher-level entities, such as subsystems, projects, and libraries, which have no analogue in most conventional programming languages, exist as well. Landscape entities are interconnected through explicitly defined relations. There are Software Landscape relations between language-level units, such as *imports*, *exports*, and *inherits*, as well as other relations, such as *part of* (*e.g.*, entity *part of* subsystem), which may not be expressible in programming languages.

The major benefit of the Software Landscape approach lies in making a large amount of software development information easily accessible to the developer, with a uniform visual mechanism that supports browsing at various levels of granularity.

## 2.2 ConceptBase: A Knowledge Base Management System

ConceptBase [9] is an implementation of the Telos [12] knowledge representation language. In the context of our research experiment, Telos has been used for specifying the Software Landscape rules of well-formedness.

The specification of the Software Landscape well-formedness rules was made using deductive rules and integrity constraints supported by Telos. These rules comprise a Telos schema

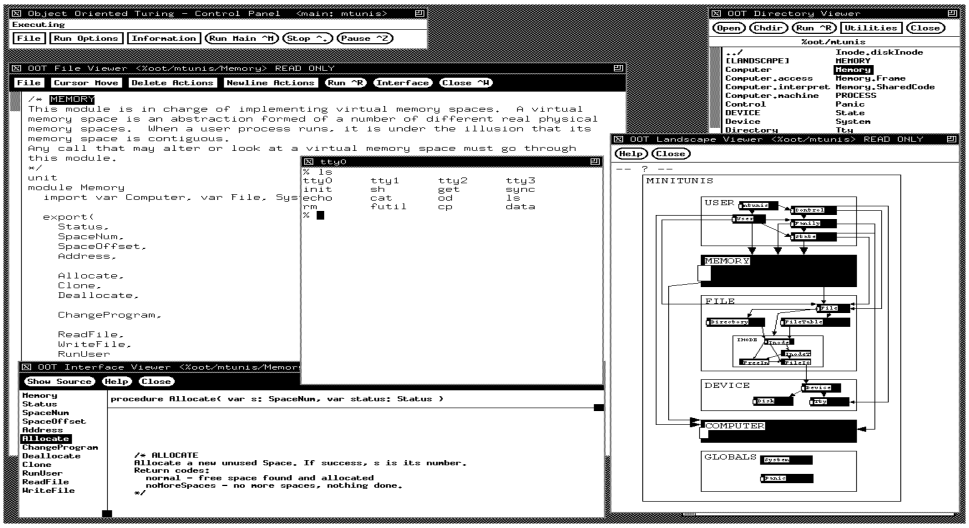


Figure 1: **OOT Snapshot.** This is a screen dump of OOT executing the MiniTunis operating system. At the top left corner is the **Control Panel**. Immediately beneath it is an instance of the OOT **File Viewer** (editor); it contains the source code for MiniTunis MEMORY module. Beneath the **File Viewer** is the **Interface Viewer**, which shows the signatures and comments associated with the interfaces of the MEMORY module. The OOT **Directory Viewer** is at the top right corner; it contains a list of files that make up the MiniTunis operating system. Below the Directory Viewer is the **Software Landscape Viewer**, which graphically depicts the structure of MiniTunis. The central window, labelled `tty0`, is a run-time window that shows the output generated by running a MiniTunis process.

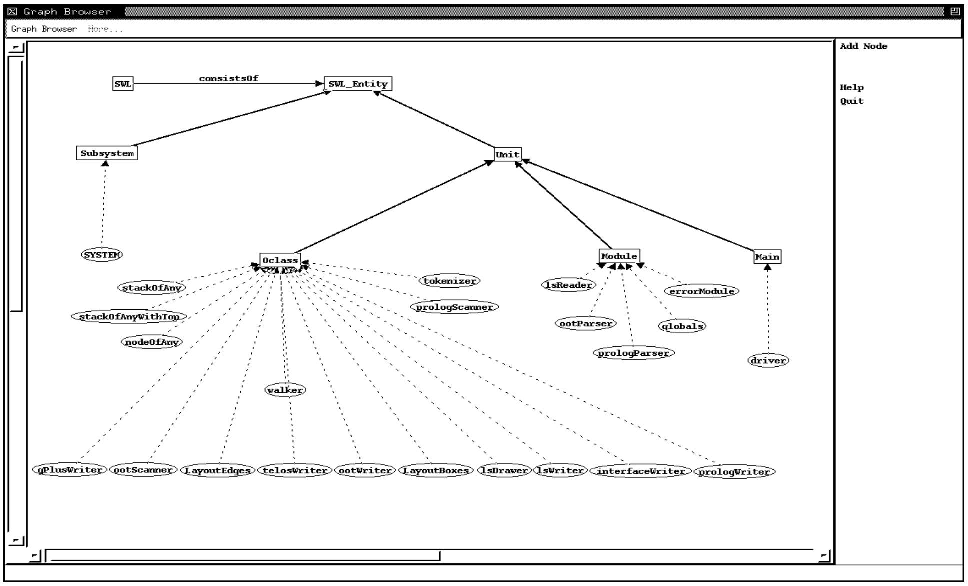


Figure 2: **ConceptBase Snapshot.** This is a screen dump of ConceptBase as it displays a graphical representation of the Telos schema for Software Landscapes along with an instance which represents the Software Landscape of the Star system. The labelled boxes and solid lines constitute the schema. In the schema, we see that a Software Landscape (SWL) consists of Software Landscape entities (SWL_Entities) which can be either Subsystems or OOT units. OOT units can be either classes (Oclass), modules, or main programs. The deductive inference rules and integrity constraints are not shown in the diagram but are expressed as Telos logic formulas. The labelled ovals and dotted lines describe the Software Landscape instance shown in Figure 5.

that precisely describes the general properties of Software Landscapes. When a particular Software Landscape is specified in Telos, it becomes an instance of the Telos schema. The developer can check the well-formedness of that instance by using ConceptBase to validate it against the schema.

Figure 2 is a snapshot from a ConceptBase display. It shows a graphical description of the Software Landscape schema augmented by an instance describing the Software Landscape shown in Figure 5. This instance describes the architecture of the Star tool integration mechanism.

### 2.3 Hy$^+$: A Data Visualization System

$Hy^+$ [4] is a data visualization system that supports the GraphLog [3] visual query language. GraphLog can be used for visualizing and querying data that exhibit graphical structures. This information is represented using graph-like presentations referred to as *hygraphs*. Hygraph nodes, edges, and boxes (called "blobs") are labelled and coloured to convey information.

An interesting characteristic of the $Hy^+$ system is that not only is the information represented visually as a hygraph, but the queries on the hygraphs are hygraphs themselves. Similar to relational databases, hygraphs are closed with respect to the query operation; that is, the result of a query on a hygraph is a new hygraph.

Although $Hy^+$ is designed to represent many kinds of structured information, we have used it to represent software structures that exist in Software Landscapes, as shown in Figure 3. In our SDE, OOT's Software Landscape viewer is used in conjunction with the $Hy^+$ system: the former provides a visual notation for large-scale programming as well as a mechanism for examining system architectures at various levels of granularity; the latter complements the Software Landscape by providing a query mechanism.

With the querying capabilities of $Hy^+$, developers can view representations of a system from perspectives other than those offered by the Software Landscape. For example, GraphLog queries can be used to restrict the amount of information shown in a Landscape, or define new relationships between Landscape entities such as indirect (transitive) export, import, and so on.

This completes the description of the three independent tools that comprise our SDE. Next, we describe the mechanism of integrating the tools into a coherent SDE.

## 3 The Star Approach to Tool Integration

Brown et al. [2] identify three kinds of tool integration:

- *User Interface Integration:* Tools have a common "look and feel" at a conceptual level, or share a common set of user interface routines (*i.e.,* Motif, OpenLook) at a physical level to ensure some degree of consistency.
- *Control Integration:* Tools are integrated via a message passing system (*i.e.,* SoftBench) in which all tools interact by sending messages to a server which rebroadcasts them selectively to only those tools that have expressed an interest in them.
- *Data Integration:* Tools are integrated either via a common data repository, such as the one for PCTE [5], or via some common data exchange format that is processable by all of the tools, such as the CASE data interchange format CDIF.

Star, although a data integration mechanism, does not rely on a repository or a common exchange format. The Star data integration approach integrates existing tools without modifying them (*a posteriori* [6]), and hence cannot guarantee user interface or control integration. The integration is based on a mechanism that translates the persistent representation used by each tool into the representations used by any other tool within the environment.

Each of the tools described in Section 2 uses text files to store its information persistently. These text files are depicted in Figure 4 as

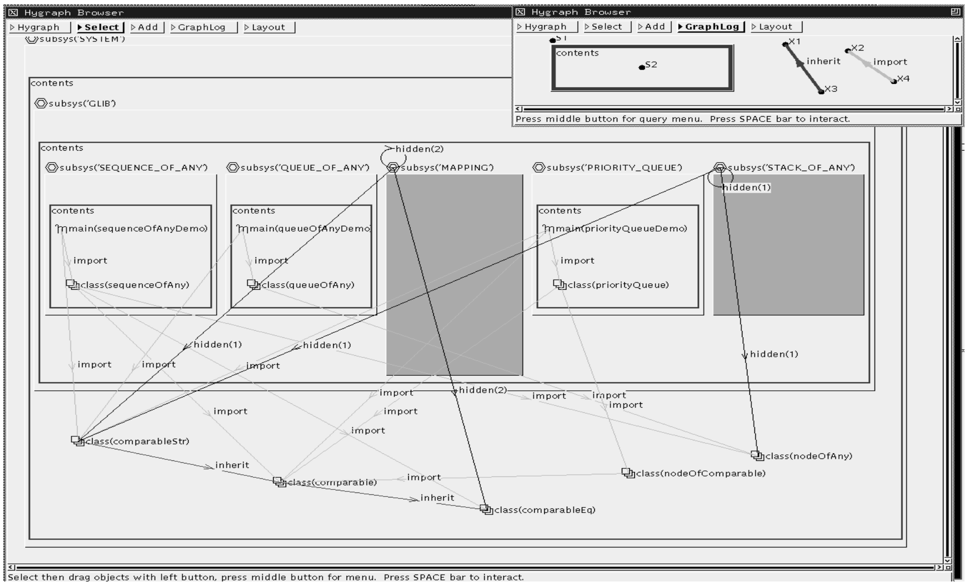


Figure 3: **Hy$^+$ Snapshot.** The small window (upper right) displays a hygraph query consisting of three parts. The left part indicates that any two entities *S1* and *S2* related via the *contents* relation in a Software Landscape should be displayed in the result hygraph as a box *S1* containing entity *S2*. The other two parts of the query indicate that Software Landscape *inherit* and *import* relationships are to be displayed as distinct coloured arrows (colour not shown in this diagram). The larger window shows the hygraph resulting from the query. The information depicted in the hygraph was automatically extracted, via Star, from the Software Landscape of an OOT library supporting data structures.

cylinders; the tools that use them are shown as rectangles. The OOT programming system stores its information as source files, each of which may have a corresponding interface file used by the Interface viewer. Software Landscapes are stored as text files that maintain the names, shapes, and positions of all entities in the diagram. The $Hy^+$ system can store hygraphs in Prolog fact files. Finally, ConceptBase can store information as Telos code in text files.

We devised a system, called Star, that translates the persistent notations from each tool to each other tool to accommodate integration. We picked the intermediate data representation to resemble that of the dominant tool, which is the Software Landscape viewer. We now describe the architecture and implementation of Star.

## 3.1 Implementation of Star

Star is written in OOT and its implementation consists of a set of parsers and a set of code generators. The parsers translate a tool's notation, such as OOT code, into a data structure which is kept in main memory. The code generators then traverse the data structure and generate text for another tool's notation, such as Software Landscapes.

The data structure is designed so that only two components, a parser and a code generator, need to be implemented to add a new tool into Star.

In Figure 5 we show a Software Landscape description of the Star implementation. At the top of the diagram is the *driver* module which comprises the main program and the user interface. The parsers are on the left side of the diagram, and the code generators are on the right. Notice that each code generator inherits from a class called *walker*, which is the common code for traversing the data structure created by the parsers.

The power of object-oriented programming greatly simplifies writing the parsers and code generators, since commonalities are factored out into abstract superclasses such as *walker*.

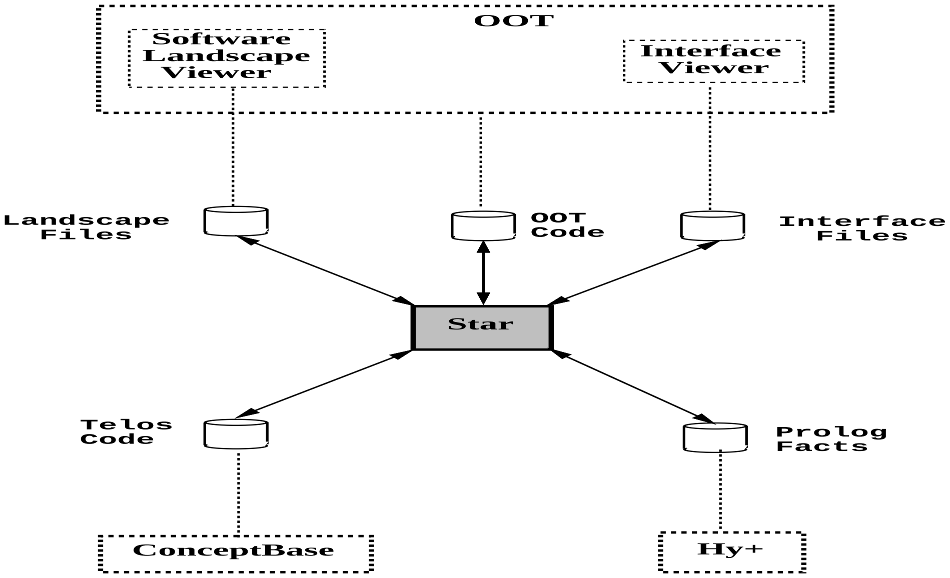


Figure 4: **Architecture of Star.** The independent tools are represented by rectangles. Shown at the top is the OOT system, which contains two tools for large-scale programming: the Software Landscape viewer and the Interface viewer. The bottom two rectangles represent the ConceptBase and $Hy^+$ systems, respectively. The persistent representation for each of these tools is indicated by cylindrical figures symbolizing files. The dotted lines between tools and files denote which tools use which files. Finally, the shaded tool in the centre is the Star system; it translates any of the persistent representations into any other to achieve tool integration.

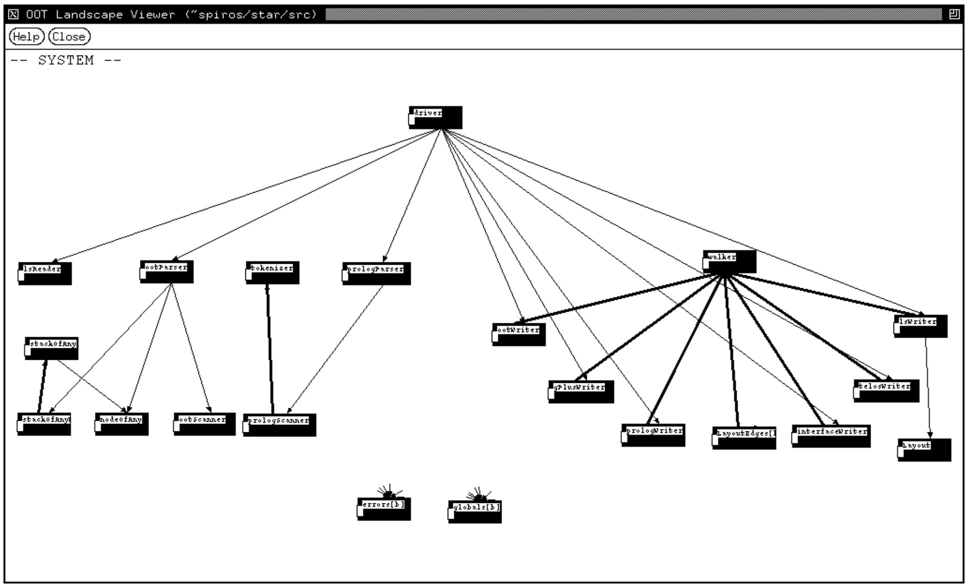


Figure 5: **Implementation of Star.** The Software Landscape is used to describe the implementation of the Star system.

# 4 Result: A Loosely Integrated SDE

By integrating the three tools described in Section 2 using the Star technique of Section 3, we quickly built a prototype SDE. This environment allows developers to view and query a software system at varying levels of granularity, from source code to unit interfaces to system architectures.

At the coarsest level of granularity, the system allows the user to extract the structure of "legacy" code as a Software Landscape and to view and navigate through it. The user can propose new architectural structuring as an extension to the Software Landscape, such as the imposition of subsystems upon existing module interconnections.

Using the ConceptBase portion of the system, the user can validate that the newly proposed structure is well formed. Queries on the structure of the system, such as determining if there are modules that are never accessed or finding the names of all modules that transitively depend on a given module, can be formulated and answered visually using the facilities of $Hy^+$.

At a finer level of granularity, the user has immediate access to the descriptions of the interfaces of each language level module, as extracted by the Star parsers and as viewed using the Interface viewer. At a finer level yet, at the level of source code, the OOT programming system supports proficient file viewing and searching.

This set of facilities to support software development has been provided by having Star translate between the various persistent data representations in a way that is largely transparent to the user. All tools, including Star, run as a separate process. Star has a simple user interface that prompts the user to enter the source file name and its language type (OOT, $Hy^+$, ConceptBase). It then parses the source and builds the in-core data structure. Subsequently, Star prompts the user to specify the language type of the source to be generated. Upon doing so, Star traverses the data structure and emits code for that language, which can be processed by its corresponding tool.

# 5 Conclusions and Future Research

In this paper we showed how it is possible to integrate a set of existing independent tools, simply and economically, to produce an SDE that supports multiple views of software artifacts.

Work is underway to extend the programming language support provided by our SDE by adding a $C$ language component to the set of translators provided by the Star system. This will allow $C$ developers to extract Landscape views of their code which can be queried using $Hy^+$ and checked for well-formedness using ConceptBase. Another aspect of Star underway is its capability for automatically laying-out Landscapes. Currently, Star uses the Sugiyama [14] layout algorithm to produce layouts of Landscapes, but other algorithms [1] might also be considered.

# About the Author

Spiros Mancoridis is a Ph.D. candidate at the University of Toronto and is member of the IBM Centre for Advanced Studies. His research interests include Software Engineering, Software Development Environments, and Software Visualization. His supervisor is Professor Richard C. Holt. Spiros Mancoridis can be reached via e-mail at *spiros@turing.utoronto.ca*.

# References

[1] 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 `/pub/papers/compgeo/gdbiblio.ps.Z`), November 1993.

[2] Brown, A. W., Earl, A. N., and McDermid, J. A. *Software Engineering Environments Automated Support for Software Engineering.* The McGraw-Hill In-

ternational Series in Software Engineering, 1992.

[3] Consens, M. P., and Mendelzon, A. O. Expressing Structural Hypertext Queries in GraphLog. In *ACM SIGCHI Hypertext '89 Conference* (1989), pp. 269–292.

[4] Consens, M. P., and Mendelzon, A. O. $Hy^{+}$: A Hygraph-based Query and Visualization System. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Washington, DC, May 1993).

[5] 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.

[6] Gohner, P. Building IPSEs by Combining Heterogeneous CASE Tools. In *Proceedings of the European Symposium on Software Development Environments and CASE Technology* (Königswinter, Germany, 1991), Springer-Verlag, pp. 18–30.

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

[8] Holt, R. C., Penny, D. A., and Mancoridis, S. Multicolour Programming and Metamorphic Programming: Object Oriented Programming-in-the-Large. In *Proceedings of the 1992 IBM CASCON Conference* (November 1992), pp. 43–58.

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

[10] Mancoridis, S., Holt, R. C., and Penny, D. A. A Conceptual Framework for Software Development. In *Proceedings of the Twenty-First ACM Computer Science Conference* (February 1993), pp. 74–80.

[11] 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.

[12] Mylopoulos, J. Conceptual Modeling and Telos. Tech. Rep. DKBS-TR-91-3, University of Toronto, November 1991.

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

[14] 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), pp. 109–125.