Automated Support for Development and Evolution of Complex Software Systems

Jack C. Wileden

COINS Technical Report 90–115
November 1990

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This paper to appear in
Proceedings of the NATO Symposium on Military Information Systems Engineering
held in Malvern, England May 1990

Abstract

The demand for ever greater levels of reliability in ever more complex software systems, especially those that are highly concurrent, distributed, or subject to stringent real-time constraints, demands increasingly powerful automated support for software developers. For the last several years, our work, in collaboration with a number of our colleagues including the other members of the Arcadia consortium, has been directed toward the development of advanced software environment and tool technology that will provide the necessary automated support for software developers. Two major foci of our work have been 1) the object management capabilities needed in a basic integrating infrastructure for advanced software environments and 2) analysis tools applicable to concurrent, distributed and real-time software. In this paper, we summarize our work in these areas, indicate its connections to the Arcadia project, and suggest future directions for efforts aimed at developing advanced software environment and tool technology.

Research supported in part by National Science Foundation grant CCR-8806970 and Office of Naval Research grant N00014-89-J-1064 and National Science Foundation grant CCR-8704478 with cooperation from Defense Advanced Research Projects Agency (ARPA order 6104).

AUTOMATED SUPPORT FOR DEVELOPMENT AND EVOLUTION OF COMPLEX SOFTWARE SYSTEMS

Jack C. Wileden*

Contents

* Associate Professor of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003 USA

1 INTRODUCTION

The demand for ever greater levels of reliability in ever more complex software systems, especially those that are highly concurrent, distributed, or subject to stringent real-time constraints, demands increasingly powerful automated support for software developers. For the last several years, our work, in collaboration with a number of colleagues including the other members of the Arcadia consortium, has been directed toward the development of advanced software environment and tool technology that will provide the necessary automated support for software developers. Two major foci of our work have been 1) the object management capabilities needed in a basic integrating infrastructure for advanced software environments and 2) analysis tools applicable to concurrent, distributed and real-time software.

In the area of object management capabilities for environments, we have developed a variety of type definition models and mechanisms. These have been specifically tailored for use in describing the components of software environments and also the components of software systems that could be developed using those environments. We have also developed and implemented an approach to adding persistence as an orthogonal property of typed objects. We believe that this approach to persistence provides the basis for moving from environments based on files to environments organized around a space of persistent typed objects. Finally we have developed the Specification Level Interoperability approach to supporting interoperation of software written in several different languages. We have produced an initial prototype implementation of this approach to interoperability that demonstrates its effectiveness by integrating a collection of software development tools written in Ada and Lisp.

In the area of analysis tools applicable to concurrent, distributed and real-time software, we have developed, and built a prototype toolset supporting, the Constrained Expression approach. Constrained expressions are a language-independent, event-based, closed-form representation for behavior of systems. We believe that this representation is particularly appropriate for describing concurrent, distributed or real-time system behaviors. Our current prototype toolset supports the analysis of concurrent systems described in an Ada-like design language. The results of our initial experiments with the toolset have been very encouraging. These results indicate that the constrained expression approach has the potential to be of practical use in analyzing realistic problems in concurrent or distributed software design. We have also carried out an initial experiment in analyzing some timing properties of a concurrent system. Our success in this experiment has led us to continue investigating the applicability of the constrained expression approach and tools to real-time system problems.

In the remainder of this paper, we give a brief overview of the structure and goals of the Arcadia project, summarize our own work in the areas of object management and analysis tools for concurrent software, and suggest future directions for efforts aimed at developing advanced software environment and tool technology.

2 ARCADIA AND SDL

The Arcadia project [15] is a collaborative software environment research program encompassing groups at several universities and industrial organizations, including the Software Development Laboratory (SDL) at the University of Massachusetts. The objective of Arcadia is to develop advanced software environment technology and to demonstrate this technology

through prototype environments. The initial Arcadia environment prototypes are being built primarily in Ada and targetted primarily to support Ada software development.

The principal research areas being addressed within the Arcadia project are environment architecture, user interface management, support for process definition and actualization, object management, support for measurement and evaluation, analysis techniques and tools, and language processing tools. Various researchers at various of the Arcadia sites are involved in working on each of these areas. Readers interested in a more comprehensive overview of Arcadia research in these two areas or in other areas of Arcadia research are referred to [15] and the papers cited therein. In the present paper, we summarize only the work on object management and analysis tools that we have carried out in collaboration with colleagues both inside and outside of SDL and the Arcadia consortium.

3 OBJECT MANAGEMENT

Extensibility, integration and interoperability are three important goals for Arcadia environments. Much of our recent effort in SDL has been directed toward the definition and implementation of support for the object management capabilities that we believe to be crucial for building integrated and extensible environments supporting tool interoperability. One focus of this research has been type models for (persistent) object management in environments. Another primary focus of the research has been on implementing persistent objects in the context of strong abstract typing. Our final focus has been on approaches to supporting tool interoperability, environment extensibility and integration. We briefly describe these efforts in the following subsections.

3.1 Type Models

One component of our work on typing support for environment developers has been the definition of a type model that we call OROS [13]. (We distinguish between *type system*, a specific collection of types developed for use in some application (such as a particular software environment or tool), and *type model*, a framework or mechanism for defining type systems.) The OROS type model is intended to permit environment builders and users to define types for environment components, such as tools or process programs, as well as types for software product components, such as requirements, designs, code, plans, and so forth. We believe that such pervasive typing can play a central role in improving the organization, increasing the reliability and facilitating the evolution of environments and the software systems that they are used to build.

The OROS type model supports both the definition of types and the determination of inter-type relationships. Our choices of the primitive types, type definition scheme and type constructors for OROS reflect our views concerning the fundamental kinds of entities that make up a software environment, the equally important roles of *relationships* and *operations* in defining the types of those entities, and the need for precise, powerful and flexible specification of inter-type relationships. We are currently experimenting with the application of these facets of the OROS model to the description of various environment components to assess the accuracy of our views and the efficacy of the model. We are also designing prototype implementations of OROS. Finally, as described in section 3.3, we are employing a subset of OROS in a prototype

implementation of our approach to supporting interoperability. Through experimenting with such prototypes and refining OROS we expect to produce a specification of the type modelling capabilities needed in advanced software development environments such as Arcadia.

3.2 Persistence

We believe that the availability of a persistent object store, smoothly integrated into the language(s) used by environment and tool builders, will dramatically simplify the building of environments. Our research on persistence has been directed toward 1) identifying an appropriate set of abstractions through which environment designers and tool builders can manipulate persistent objects, and 2) exploring implementation strategies for persistence.

The PGRAPHITE system [17] is our currently operational prototype of a persistent object capability. This system is a preprocessor that accepts definitions of abstract graph data types, specified in the Graph Definition Language (GDL) [7], and produces an Ada implementation of the specified graph types incorporating orthogonal persistence. PGRAPHITE provides environment designers and tool builders with three sets of abstractions for manipulating persistent objects, namely *persistent object*, *persistent store* and *graph* abstractions. Our persistent object abstraction augments the operations available on any type with operations that can make individual objects of that type become persistent. Hence the persistence property is orthogonal to any other properties of a type, persistence can be controlled on an instance by instance basis, and tools interact with a persistent store containing persistently typed objects, rather than with a file system containing only one persistent type (namely file). Under the PGRAPHITE persistent store abstraction, tools can access persistent stores, which are called repositories, only during a *session*, and a tool must explicitly indicate the beginning and end of sessions. Sessions provide a basis for concurrency management and hence support sharing of persistent graph nodes, both among tools and between two or more graphs.

Our implementation strategy has several interesting features. In order to preserve abstract typing and information hiding, object classes manage their own persistence in PGRAPHITE. Efficiency of both memory utilization and I/O traffic is increased through fault-driven retrieval of objects. The PGRAPHITE processor automatically generates Ada implementations of abstract types and repository managers. While this automatic generation capability is currently available only for abstract graph types describable in GDL, the approach is completely general and we have manually applied it to a wide range of types. We have also ported our PGRAPHITE implementation to several different underlying storage managers, including Ada Direct_IO and the Mneme system [12]. This porting is facilitated by the standardized Storage Manager Interface that is part of the PGRAPHITE implementation architecture.

Through use of the PGRAPHITE system, both within SDL and at several other sites, we are exploring several important issues concerning persistent typed object management in environments. We are currently working to extend PGRAPHITE by automating the generation of additional types, and by adding support for concurrency, version control and garbage collection. We are also investigating "principled" approaches to limiting the extent of persistence, which will complement the reachability-based definition of extent of persistence that is provided by PGRAPHITE [19].

3.3 Interoperability

There is an increasing need and desire to develop systems, and especially environments, by combining components that are written in different languages and/or that are run on different kinds of machines. Success at this depends in large part on the *interoperability* of the components—that is, the ability of the components to communicate and work together despite their differing backgrounds. While most previous approaches to interoperability have provided support at the representation level, we are pursuing an approach that will provide support at the specification level. We have developed a model of Specification Level Interoperability (SLI) [18] that consists of four components: 1) a *unified type model*, which is a notation for describing the entities to be shared by interoperating programs; 2) *language bindings*, which connect the type models of the languages to the unified type model; 3) *underlying representations and implementations*, which realize the types used by the different interoperating programs; and 4) *automated assistance*, which generally eases the task of combining components into an interoperable whole. To demonstrate and investigate SLI, we have created a prototype realization of the approach and applied it to achieving interoperability of several components of the constrained expression toolset described in section 4.2.

The unified type model (UTM) concept is central to the SLI approach. The intent is that a UTM should serve as a basis for tool cooperation within a richly structured collection of environment components. Integration will be enhanced by permitting the tools to share and exploit the rich structure of those components. This is in direct contrast to the Unix model, for example, a representation level approach to interoperability that forces tools to interact at the level of byte streams. The richer structure supported by a UTM will permit consistency checking (e.g., type checking) and will free tools from the necessity of explicitly translating (parsing and/or unparsing) inputs and outputs. A UTM can be viewed as a semi-strong coupling of environment components - stronger than typeless byte streams, but weaker than a single fixed type system. We believe that this intermediate position is the key to simultaneously attaining interoperability, integration and extensibility.

The UTM included in our initial prototype realization of SLI represents an attempt to be compatible with a variety of existing or proposed type models and type systems and is also intended to be appropriate for short term practical use in Arcadia prototypes. This initial UTM is based upon a simple subset of OROS concepts. It consists of a set of type definition primitives, a set of relationships or constructor functions, a set of "special" types, and some semantics for manipulation of instances. We have successfully used this UTM in our prototype SLI realization to describe and support automatic generation of multilingual implementations of an object type that is central to the constrained expression toolset.

4 ANALYSIS TOOLS FOR CONCURRENT SOFTWARE

A wide variety of techniques have been proposed for analyzing the behavior of concurrent software systems. These differ in their underlying models of concurrent computation, in the questions about behavior they attempt to answer, and in the stages of the software development process in which they are applied. It is, of course, unlikely that any single approach to analysis can possibly meet all the needs of software developers throughout the development process. Therefore, the goals of the Arcadia project include developing, and facilitating the integration

of, various approaches to analyzing the behavior of concurrent software systems.

In the following subsections we briefly describe one approach to analysis, called the Constrained Expression approach, that has been developed in collaboration with colleagues outside the Arcadia consortium, but is a candidate for inclusion in such an integrated collection of analysis capabilities. We outline the approach itself, a prototype toolset supporting that approach, and the results of some preliminary experiments with the toolset.

4.1 Constrained Expressions

In the constrained expression approach to analysis of concurrent systems, the system descriptions produced during software development (e.g., designs in some design notation) are translated into formal representations, called *constrained expression representations*, to which a variety of analysis methods are then applied. This approach allows developers to work in the design notations and implementation languages most appropriate to their tasks. Rigorous analysis is based on the constrained expression representations that are mechanically generated from the system descriptions created by software developers.

This subsection contains a brief overview of the constrained expression formalism. A detailed and rigorous presentation is given in [8], and a less formal treatment presenting the motivation for many of the features of the formalism appears in [4]. The use of constrained expressions with a variety of development notations is illustrated in [4] and [10].

The constrained expression formalism treats the behaviors of a concurrent system as sequences of events. These events can be of arbitrary complexity, depending on the system characteristics of interest and the level of system description under consideration. Associating an *event symbol* to each event, we can regard each possible behavior of the system as a string over the alphabet of event symbols.

We use interleaving to represent concurrency. Thus, a string representing a possible behavior of a system that consists of several concurrently executing components is obtained by interleaving strings representing the behaviors of the components. The events themselves are assumed to be atomic and indivisible. "Events" that are to be explicitly regarded as overlapping in time are represented by treating their initiation and termination as distinct atomic events.

The set of strings representing behaviors of a particular concurrent system is obtained by a two-step process. First, a regular expression, called the *system expression*, is derived from a description of the system in some notation such as a design or programming language. The language of the system expression includes strings representing all possible behaviors of the system. It may, however, also include strings that do not represent possible behaviors, as the system expression does not encode the full semantics of the system description. This language is then "filtered" to remove such strings, using other expressions, called *constraints*, which are also derived from the original system description. A string survives this filtering process if its projections on the alphabets of the constraints lie in the languages of the constraints. The constraints (which need not be regular) enforce those aspects of the semantics of the design or programming language, such as the appropriate synchronization of rendezvous between different tasks or the consistent use of data, that are not captured in the system expression. The reasons for this two-step process, which might not seem as straightforward as generating behaviors directly from a single expression, are discussed in [10].

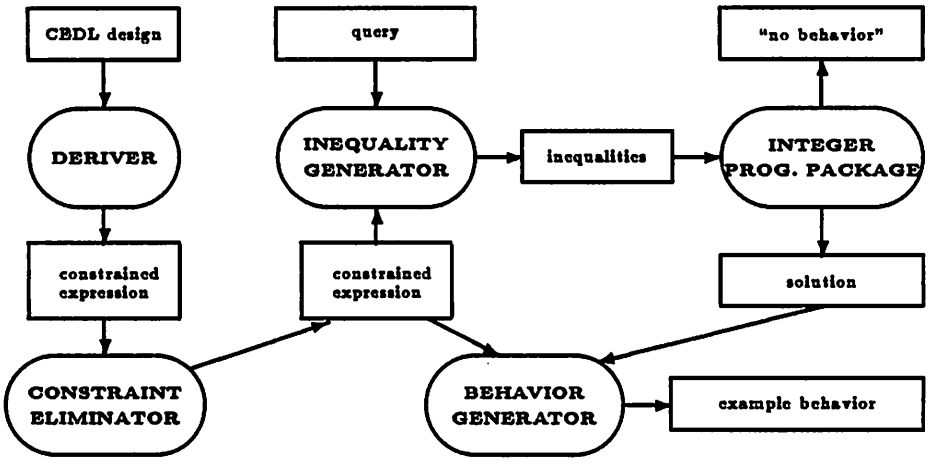

Figure 1: Diagram of Constrained Expression Toolset

Our main constrained expression analysis techniques require that questions about the behavior of a concurrent system be formulated in terms of whether a particular event symbol, or pattern of event symbols, occurs in a string representing a possible behavior of the system. For example, questions about whether the system can deadlock might be phrased in terms of the occurrence of symbols representing the starvation of component processes of the system.

Starting from the assumption that the specified symbol, or pattern of symbols, does occur in such a string, we use the form of the system expression and the constraints to generate inequalities involving the numbers of occurrences of various event symbols in segments of the string. If the system of inequalities thus generated is inconsistent, the original assumption is incorrect and the specified symbol or pattern of symbols does not occur in a string corresponding to a behavior of the system. If the inequalities are consistent, we use them in attempting to construct a string containing the specified pattern.

4.2 The Constrained Expression Tools

After manually applying the constrained expression analysis techniques to a number of small examples with encouraging results (e.g., [1], [4], [5], [16]), we began to construct prototype tools automating various aspects of the analysis. The prototype toolset (see Figure 1) consists of five major components: a *deriver* that produces constrained expression representations from concurrent system designs in a particular design language; a *constraint eliminator* that replaces a constrained expression with an equivalent one involving fewer constraints; an *inequality generator* that generates a system of inequalities from the constrained expression representation of a concurrent system; an *integer programming package* for determining whether this system of inequalities is consistent or inconsistent, and, if the system is consistent, for finding a solution with appropriate properties; and a *behavior generator* that uses the constrained expression and the solution found by the integer programming package (when the inequalities are consistent) to produce a string of event symbols corresponding to a system behavior with the desired properties. The organization of the toolset is illustrated in the figure.

The current toolset is intended for use with designs written in the Ada-based design language CEDL (Constrained Expression Design Language) [9]. CEDL focuses on the expression

of communication and synchronization among the tasks in a distributed system, and language features not related to concurrency are kept to a minimum. Thus, for example, data types are limited, but most of the Ada control-flow constructs have correspondents in CEDL. The deriver in the CEDL toolset is written in Ada, and was developed using Arcadia-produced versions of standard compiler construction tools plus PGRAPHITE. The constraint eliminator, the inequality generator, and the behavior generator are all written in Common Lisp, while the integer programming component, which is built on top of the MINOS optimization package [14], is implemented in FORTRAN. As mentioned in the previous section, we have experimented with using our SLI techniques and prototype tools to implement interoperability between the Ada and Common Lisp components of the toolset.

4.3 Experiments with the Toolset

We have begun to use the prototype toolset in the analysis of concurrent systems. The preliminary experiments reported here represent an initial attempt to determine the practical limitations of automated support for the constrained expression approach to analysis. A number of variations of four different systems have been analyzed. First, a standard formulation of the dining philosophers problem provides a basis for comparison with other analysis techniques because of its widespread use as a benchmark problem. The addition of a host task controlling entry to the dining room indicates how the introduction of intra-task dataflow affects tool performance. The readers/writers problem requires analysis of more complex data flow patterns. Finally, we analyze an automated gas station example in which both the synchronization patterns and intra-task dataflow are relatively complex. Varying the number of philosophers in the dining philosophers problems indicates how increasing the number of tasks in the system being analyzed affects tool performance.

The table in Figure 2 gives CPU times for the application of the components of the toolset to these systems. All times are in CPU seconds on a Sun 3/60, except for the deriver time in the DPH-14 case, where memory limitations forced us to run on a 3/260. The first section of the table, with systems labeled DP-n, gives data for analyses of the standard dining philosophers problem involving n philosophers. A bug in the behavior generator prevents us from completing the analysis of the DP-10 case. The next section, with systems labeled DPH-n, gives data for analyses of versions of the problem with n philosophers and a host task that prevents deadlock by limiting the number of philosophers in the dining room. The third section gives the results for two versions, one incorrect and one correct, of the readers/writers example [6]. In the correct version, the analysis determines whether an error flag, representing a violation of the appropriate mutual exclusion, is ever set. The symbol representing the setting of this flag is correctly eliminated by the constraint eliminator; once this is recognized by the inequality generator, no further analysis is necessary. Times for analyses of two versions of Helmbold and Luckham's automated gas station [11], one with a potential deadlock and the other without, are given in the last section. More information on these systems and the analyses (but with less current performance data) is provided in [2].

These initial experiments with the prototype constrained expression toolset are encouraging. The toolset provides complete automated analysis of a range of standard concurrent system examples. Even the prototype versions of the tools are efficient enough to be useful to software developers on examples of moderate size. Unlike the standard approaches to concur-

system	deriver	constraint eliminator	inequality generator	int. prog. (IMINOS)	behavior generator	total CPU time
DP-3	74	1	9	2	19	105
DP-4	82	2	11	3	26	124
DP-5	94	3	14	3	32	146
DP-6	109	4	17	4	38	172
DP-8	142	7	24	5	54	232
DP-10	177	11	30	7		
DPH-3	123	6	16	3	—	148
DPH-4	133	9	22	4	—	168
DPH-5	152	14	30	6	—	202
DPH-6	174	19	38	7	—	238
DPH-8	207	31	57	11	—	306
DPH-10	250	49	82	30	—	411
DPH-14	233	101	135	57	—	526
RW-I	43	6	9	7	124	189
RW-C	63	10	2	—	—	75
GAS-I	75	21	30	13	721	860
GAS-C	76	16	23	13	—	128

Figure 2: CPU times, in seconds, for the constrained expression tools.

rency analysis, which are based on a reachability tree construction that grows exponentially with the number of concurrent tasks being analyzed, our toolset does not appear to suffer from exponential performance degradation as problem size increases. Furthermore, earlier experiments show that the constrained expression approach can detect a variety of errors and can be used with a broad range of design notations and programming languages.

We are currently reimplementing the behavior generator, to remove the bug that we have encountered in the DP-10 case, to enable it to use all the information provided by the solution to the system of inequalities, and to add some additional functionality. We expect that significant improvements in its performance will result from the use of more information from the solution.

While improving the prototype toolset, we have also begun to explore additional applications for constrained expression analysis, some of which may lead to enhancements to the underlying formalism and further modifications to the tools. In particular, we have begun to study the application of the constrained expression approach to various scheduling and real-time problems [3].

5 SUMMARY AND CONCLUSIONS

In this paper we have provided brief overviews of our work on two important aspects of automated support for development and evolution of complex software systems. Our work on type models, persistence and interoperability is representative of the directions that object management capabilities for environments are likely to take over the next decade. Our constrained expression toolset is an example of the automated analysis techniques that will

be required if such environments are to adequately support development and evolution of concurrent, real-time or other classes of complex software.

Perhaps the most fundamental aspects of our object management work are its emphasis on strong, abstract typing, and its tendency toward incorporation of object management capabilities (e.g., persistence) into the language in which tools are implemented, as opposed to the current practice of providing such capabilities via a database or file system. We believe that these directions will be important factors in increasing integration, extensibility and interoperability in the next generation of software development environments.

While we believe that the results of our experiments with our prototype constrained expression toolset are significant, we see the empirical approach to evaluation of proposed approaches to software analysis that the experiments represent as equally important. Only by carrying out such empirical evaluations, preferably using a suite of standard examples, can the relative strengths and weaknesses of various proposed approaches be determined. Assembling collections of tools with known, complementary capabilities on top of infrastructures that facilitate their integration and interoperability is the most promising approach to providing automated support for development and evolution of complex software systems.

Acknowledgments

The work on object management described here was supported in part by NSF grant CCR-87-04478 with cooperation from DARPA (ARPA order 6104). Alexander Wolf has played a major role in all of the object management research described in this paper. Lori Clarke, William Rosenblatt and Peri Tarr have also made important contributions to the SDL object management work reported here. Many others of our Arcadia colleagues have provided valuable input to this work.

The work on constrained expression analysis described here was supported in part by NSF grant CCR-8806970 and by ONR grant N00014-89-J-1064. George Avrunin and Laura Dillon have been centrally involved in the development of the constrained expression approach to analysis of concurrent software systems. Susan Avery, Ugo Buy, James Corbett, Michael Greenberg, RenHung Hwang, and George Walden have played important roles in the development of the toolset and the experimental evaluation reported in this paper.

REFERENCES

[1] G. S. Avrunin. Experiments in constrained expression analysis. Technical Report 87-125, Department of Computer and Information Science, University of Massachusetts, Amherst, November 1987.

[2] G. S. Avrunin, L. K. Dillon, and J. C. Wileden. Experiments with Automated Constrained Expression Analysis of Concurrent Software Systems. In *Proceedings TAV3-SIGSOFT89: Third Testing, Analysis and Verification Symposium*, pages 124–130, December 1989.

[3] G. S. Avrunin, L. K. Dillon, and J. C. Wileden. Constrained expression analysis of real-time systems. Technical Report 89-50, Department of Computer and Information Science, University of Massachusetts, 1989.

[4] G. S. Avrunin, L. K. Dillon, J. C. Wileden, and W. E. Riddle. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Trans. Softw. Eng.*, SE-12(2):278–292, 1986.

[5] G. S. Avrunin and J. C. Wileden. Describing and analyzing distributed software system designs. *ACM Trans. Prog. Lang. Syst.*, 7(3):380–403, July 1985.

[6] R. H. Carver and K.-C. Tai. Detection of synchronization errors in concurrent software by semantics-based analysis. Preprint, 1988.

[7] L. A. Clarke, J. C. Wileden, and A. L. Wolf. GRAPHITE: A meta-tool for Ada environment development. In *Proceedings of 2nd International Conference on Ada Applications and Environments*, pages 81–90, April 1986.

[8] L. K. Dillon. *Analysis of Distributed Systems Using Constrained Expressions.* PhD thesis, University of Massachusetts, Amherst, 1984.

[9] L. K. Dillon. Overview of the constrained expression design language. Technical Report TRCS86-21, Department of Computer Science, University of California, Santa Barbara, October 1986.

[10] L. K. Dillon, G. S. Avrunin, and J. C. Wileden. Constrained expressions: Toward broad applicability of analysis methods for distributed software systems. *ACM Trans. Prog. Lang. Syst.*, 10(3):374–402, July 1988.

[11] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[12] J. E. B. Moss and S. Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface. In *Proceeding of the Second International Workshop on Object Oriented Data Bases*, Springer-Verlag, September 1988.

[13] W. R. Rosenblatt, J. C. Wileden, and A. L. Wolf. OROS: Toward a Type Model for Software Development Environments. In *Proceedings OOPSLA'89: Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 297–304, October 1989.

[14] M. A. Saunders. MINOS system manual. Technical Report SOL 77-31, Stanford University, Department of Operations Research, 1977.

[15] R. N. Taylor, F. C. Belz, L. A. Clarke, L. J. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolf, and M. Young. Foundations for the Arcadia environment architecture. In *Proceedings SIGSOFT '88: Third Symposium on Software Development Environments*, pages 1–13, December 1988.

[16] J. C. Wileden and G. S. Avrunin. Toward automating analysis support for developers of distributed software. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 350–357. IEEE Computer Society Press, June 1988.

[17] J. C. Wileden, A. L. Wolf, C. D. Fisher, and P. L. Tarr. PGRAPHITE: An Experiment in Persistent Typed Object Management. In *Proceedings SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, December 1988.

[18] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, and P. L. Tarr. Specification level interoperability. In *Proceedings of the Twelfth International Conference on Software Engineering*, pages 74–85, March 1990.

[19] J. C. Wileden, P. L. Tarr, and L. A. Clarke. Extending and Limiting PGRAPHITE-style Persistence. Submitted.