N88-17212

Simplifying the Construction of Domain-Specific Automatic Programming Systems: The NASA Automated Software Development Workstation Project

Bradley P. Allen
Peter L. Holtzman

Inference Corporation
5300 W. Century Blvd.
Los Angeles, CA 90045

Abstract

We provide an overview of the Automated Software Development Workstation Project, an effort to explore knowledge-based approaches to increasing software productivity. The project focuses on applying the concept of domain-specific automatic programming systems (D-SAPSs) to application domains at NASA's Johnson Space Center. We describe a version of a D-SAPS developed in the Phase I of the project for the domain of Space Station momentum management, and discuss how problems encountered during its implementation have led us to concentrate our efforts on simplifying the process of building and extending such systems. We propose to do this by attacking three observed bottlenecks in the D-SAPS development process through the increased automation of the acquisition of programming knowledge and the use of an object-oriented development methodology at all stages of program design. We discuss how these ideas are being implemented in the Bauhaus, a prototype CASE workstation for D-SAPS development.

1. Increasing software productivity through domain-specific automatic programming

Software development has come under criticism as an increasingly serious bottleneck in the construction of complex automated systems. Increasing the reuse of software designs and components has been viewed as an important way to address this problem, possibly increasing productivity by an order of magnitude or more [9]. A promising approach to achieving software reusability is through *domain-specific automatic programming*.

Domain-specific automatic programming systems (D-SAPS) use application domain knowledge to automate the refinement of a program description (written in a high-level domain language) into compilable code (written in a procedural target language) [1]. D-SAPSs can be distinguished from the more traditional domain-independent automatic programming systems in that the specification of the program is in a domain-specific language accessible to an end user, rather than a formal specification language (e.g. the predicate calculus with equality). Application generators of the type used in business report generation (e.g. Focus and DBASE-II) are examples of D-SAPSs in which the refinement process is completely automatic and implemented procedurally [10]. More complex domains can be handled if the user is allowed to interact with and guide the refinement process. Prototype knowledge-based systems that support user interaction and which work for practical application domains have been successfully developed (e.g. Draco [16], ϕNIX [3], and KBEmacs [25]).

2. The Automated Software Development Workstation Project

Since the fall of 1985, Inference Corporation has been involved in an effort, sponsored by NASA's Lyndon B. Johnson Space Center, to explore the applicability of domain-specific automatic programming to NASA software development efforts. Phase I of the project focused on the development of a D-SAPS for the domain

of Space Station momentum management [19]. During Phase I, A prototype D-SAPS was constructed, comprised of:

- a components catalog of FORTRAN subroutines used in the construction of Space Station orbital simulations;
- a design catalog of programs implemented using the system;
- a interactive graphical design system using a dataflow specification language for design editing and components composition;
- code generators for component interfaces, numerical subroutines and main programs; and
- a rule-based expert that:
  - proposed refinements for unimplemented modules in the dataflow specification;
  - flagged inconsistencies at manually-specified component interfaces; and
  - suggested possible workarounds to "patch" inconsistencies (e.g. coordinate system conversion routines).

The system was implemented by hand, to serve as a model for the implementation of similar D-SAPS for other NASA domains. The functionality and performance of the prototype was adequate to demonstrate the applicability of the D-SAPS approach to software development at NASA JSC. However, reflecting on our experience in building this system led us to be in accord with other D-SAPS developers in noting that:

- "domain analysis and design is *very hard*" [16]; and
- "domain-specific systems can be quite useful within their range of application, but the range is often quite narrow" [2].

We feel that these two issues must be addressed if D-SAPSs are to play a significant role in future software development environments. Therefore, in Phase II of the project, we are attempting to address the bottlenecks in the D-SAPS development process that lead to these observations.

3. Addressing the bottlenecks in D-SAPS development

We have focused on three bottlenecks in the D-SAPS development process that were observed during the development of the Phase I system:

- developing a domain language;
- describing design refinements and constraints; and
- describing the generation of target language code from a sufficiently detailed program description.

We plan to reduce the effort spent on each of these tasks by:

- automating the programming knowledge acquisition process; and
- using an object-oriented development methodology at all stages of the program design process.

We are currently implementing a knowledge-based D-SAPS development workstation, called the Bauhaus, that will embody these two approaches. We now describe how the design of the Bauhaus addresses the perceived bottlenecks.

3.1. Automating the programming knowledge acquisition process

By structuring the design process so that the types of knowledge required are made explicit, the knowledge acquisition process can be made simpler [14], and the resulting knowledge base easier to maintain [20]. To this end we are using a problem solving architecture based that of the RIME [24] and SOAR [12] systems, implemented using the ART expert system building tool [11]. This architecture allows us to organize design knowledge into a hierarchy of *problem spaces*, representing program design tasks. Each problem space consists of a set of operators for performing the task

represented by the space. In the Bauhaus, a problem space is associated with each program description that the system has in its knowledge base; the task represented by the problem space is that of refining the program description until it is sufficiently detailed to allow a code generator to translate it into code in the target language. The design process in the Bauhaus occurs in the following way:

1. **Select the initial design problem**: the user copies and edits an initial program description from the set of program descriptions in the knowledge base using a structure editor, making it the *current description*. The initial problem space is that associated with refining this description.

2. **Propose operators**: the Bauhaus determines what operators are applicable to the current description.

3. **Choose an operator**: the Bauhaus chooses an operator from the proposed set using operator preferences and constraints associated with the problem space, and implemented as ART production rules. User interaction is requested when the system reaches an *impasse*, where either no operator is known to apply, the system is unable to derive a preference for a specific operator, or the system's preferences are inconsistent [12]. This interaction takes one of two forms:

   - the user chooses a proposed operator for the system to apply; or
   - the user edits the current program description, in which case we return to step 2.

4. **Apply the chosen operator**: the Bauhaus applies the chosen operator to the current description. The operator may:

   - select a new problem space,
   - recurse into a problem subspace,
   - refine the current description,
   - signal that the task for the problem space is complete, or
   - signal that the task cannot be successful completed.

   We then return to step 2.

The design process terminates when the top-level task of refining the initial program description is successfully completed. This occurs when the description is detailed enough to allow the generation of target language code to occur. Given this problem solving architecture, we now discuss the knowledge acquisition mechanisms used to obtain the descriptions, operators, operators preferences and constraints used in the design process.

3.1.1. Acquiring descriptions

Our representation of domain objects and operations uses a description language, implemented in ART schemata, that is similar to KRYPTON [17]. New descriptions of domain objects and operations are created from existing descriptions using the copy&edit technique espoused by Lenat in the CYC system, [13] and are placed in the appropriate location in a subsumption hierarchy through an automatic classifier [8]. This use of description copy&edit together with automatic classification reduces the effort required to extend the domain language used to describe systems, by fostering reuse of existing domain languages in the creation of new domain languages. Using a subsumption hierarchy of descriptions as the organizing framework for the representation of objects and operations supports user access for copy&edit actions through a retrieval-by-reformulation browser similar in design to ARGON [18]. Retrieval-by-reformulation will permit a naive user of the Bauhaus to find a description needed for a copy&edit action more easily than using a tradition query mechanism [23].

3.1.2. Acquiring operators, operator preferences, and constraints

When the user performs a manual edit of a description in response to an impasse, the Bauhaus will create an operator whose condition is the current description and whose action is the manual editing action. Operator preferences are acquired by recording the conditions under which a user makes a selection from a set of

operators during an impasse where no operator is preferred. Constraints are acquired when a user manually rejects the application of an operator, causing the Bauhaus to backtrack to the previous problem solving state. This type of knowledge acquisition through the observation of manual programming steps taken by the user can be characterized as a *learning apprentice* approach [15]. In this respect, the Bauhaus is similar to the VEXED VLSI design system [22].

3.2. Using an object-oriented development methodology

By using object-oriented design (OOD) [5], we can decrease the level of effort required to implement a code generator that takes a sufficiently detailed program description and produces compilable target language code. This is due to the natural correspondence between the world and its model in an object-oriented framework [7]. In the Bauhaus, the world is the set of target language software components and code templates and the model is the set of descriptions of objects and sequences of operations in an application program. Ada's language level support of abstract data types and the existence of commercially supported reusable software component libraries constructed using OOD principles [6] make it our first choice as a target language in the Bauhaus system. The mapping between the description of an program and its realization in Ada code and the generation of the main subprogram in which the program objects are scoped is straightforward. We believe that the Bauhaus could easily be extended to support other languages with similar OOD features as target languages (e.g. Smalltalk, Objective-C, or ART).

4. System status and limitations

Implementation of the Bauhaus is currently underway using ART running on a Symbolics Lisp machine under the Genera 7.1 environment. Support for Ada compilation and library management is provided by the Symbolics Ada programming environment. As of July 1987, ART-based representations for descriptions, operators, operator preferences and constraints have been designed, the problem solving architecture and basic knowledge acquisition algorithms have been designed and implemented, and the target language reusable components library has been selected. The user interface is currently under construction, and the domain analysis for the demonstration domain, orbital flight simulation, is underway. We plan to demonstrate the use of the Bauhaus in the construction of a D-SAPS for this domain in the first quarter of 1988.

In the current design of the Bauhaus, there are a number of issues relevant to D-SAPSs that we do not address:

- **Lifecycle issues**: the Bauhaus is only useful as a programming-in-the-small environment, and ignores programming-in-the-large issues (e.g. version control). These would have to be addressed in a production-quality system.

- **Persistent object bases**: the Bauhaus has no provision for saving session state in a more sophisticated manner than simply saving changes out to a text file. We are looking to work on object-oriented databases to provide an answer here [4].

- **Automated algorithm synthesis**: the Bauhaus will always reach an impasse if a programming task requires algorithm design. However, the architecture should be extensible to encompass this kind of problem solving (e.g., see the work by Steier on the Cypress-Soar and Designer-Soar algorithm design systems [21]).

5. Conclusion

There is evidence that domain-specific automatic programming is a viable approach to increasing software productivity. To make this approach a practical one, the task of building and extending D-SAPS must be made simpler. As described above, we plan to accomplish this by improving the knowledge acquisition and software engineering methodologies used in constructing D-SAPS. Our ultimate goal is a production-

quality system that could be described as an "application generator generator"; i.e., a knowledge-based environment for the construction of special-purpose systems for the generation of applications software by end-users. Such a system could be available to systems analysts and designers in a DP/MIS organization for use when an applications programming task occurs frequently enough to merit the creation of a D-SAPS.

References

1. Barstow, D. "Domain-Specific Automatic Programming". *IEEE Transactions on Software Engineering 11*, 11 (November 1985).

2. Barstow, D. Artificial Intelligence and Software Engineering. Proceedings of the 9th International Conference on Software Engineering, IEEE, March-April, 1987.

3. Barstow, D., Duffey, R., Smoliar, S., and Vestal, S. An overview of ΦNIX. Proceedings of the Second National Conference on Artificial Intelligence, AAAI, August, 1982.

4. Bernstein, P.A. Database System Support for Software Engineering. Proceedings of the 9th International Conference on Software Engineering, IEEE, March-April, 1987.

5. Booch, G. "Object-Oriented Development". *IEEE Transactions on Software Engineering 12*, 2 (February 1986).

6. Booch, G. *Software Components With Ada.* Benjamin/Cummings Publishing, 1987.

7. Bordiga, A., Greenspan, S., and Mylopoulos, J. "Knowledge Representation as the Basis for Requirements Specification". *Computer 18*, 4 (April 1985).

8. Brachman, R.J. and Levesque, H.J. The Tractability of Subsumption in Frame-Based Description Languages. Proceedings of the National Conference on Artificial Intelligence, AAAI, August, 1984.

9. Horowitz, E. and Munson, J.B. "An Expansive View of Reusable Software". *IEEE Transactions on Software Engineering 10*, 5 (September 1984).

10. Horowitz, E., Kemper, A., and Narasimhan, B. Application Generators: Ideas for Programming Language Extensions. Proceedings of ACM'84 Annual Conference: The Fifth Generation Challenge, ACM, October, 1984.

11. Inference Corporation. *ART 3.0 Reference Manual.* Inference Corporation, 1987.

12. Laird, J.E., Newell, A. and Rosenbloom, P.S. "SOAR: An Architecture for General Intelligence". *Artificial Intelligence 33*, 1 (1987).

13. Lenat, D.B., Prakash, M., and Shepherd, M. "CYC: Using Common Sense Knowledge To Overcome Brittleness and Knowledge Acquisition Bottlenecks". *AI Magazine 6*, 4 (Winter 1986).

14. Marcus, S., McDermott, J., and Wang, T. Knowledge Acquisition for Constructive Systems. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, August, 1985.

15. Mitchell, T.M., Mahadevan, S., and Steinberg, L.I. LEAP: A Learning Apprentice for VLSI Design. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, August, 1985.

16. Neighbors, J.M. "The Draco Approach to Constructing Software from Reusable Components". *IEEE Transactions on Software Engineering 10*, 5 (September 1984).

17. Patel-Schneider, P.F. Small can be Beautiful in Knowledge Representation. Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems, December, 1984.

18. Patel-Schneider, P.F., Brachman, R.J., and Levesque, H.J. ARGON: Knowledge Representation meets Information Retrieval. Proceedings of the First Conference on Applications of Artificial Intelligence, IEEE, December, 1984.

19. Prouty, D.A. and Klahr, P. Automated Software Development Workstation. Proceedings of the Conference on AI for Space Applications, NASA, November, 1986.

20. Soloway, E., Bachant, J. and Jensen, K. Assessing the Maintainability of XCON-in-RIME: Coping with the Problems of a VERY Large Rule Base. Proceedings of the National Conference on Artificial Intelligence, AAAI, July, 1987.

21. Steier, D.M., Laird, J.E., Newell, A., Rosenbloom, P.S., Flynn, R.A., Golding, A., Polk, T.A., Shivers, O.G., Unruh, A. and Yost, G.R. Varieties of Learning in Soar: 1987. Proceedings of the Fourth International Workshop on Machine Learning, June, 1987.

22. Steinberg, L.I. Design as Refinement Plus Constraint Propagation: The VEXED Experience. Proceedings of the National Conference on Artificial Intelligence, AAAI, July, 1987.

23. Tou, F.N., Williams, M.D., Fikes, R., Henderson, A., and Malone, T. RABBIT: An Intelligent Database Assistant. Proceedings of the Second National Conference on Artificial Intelligence, AAAI, August, 1982.

24. van de Brug, A., Bachant, J. and McDermott, J. "The Taming of R1". *IEEE Expert 1*, 3 (Fall 1986).

25. Waters, R.C. "The Programmer's Apprentice: A Session with KBEmacs". *IEEE Transactions on Software Engineering 11*, 11 (November 1985).