The Anatomy of a Component Generator[1]

Walid Taha & Jim Hook

{walidt, hook}@cse.ogi.edu

The Oregon Graduate Institute

In this extended abstract, we outline some essential elements of a conceptual model for a component generation system. This model is based on an extensive study of a large number of high-level program generation systems, and the significant body of related literature. We focus our attention on the architectural elements of this model, and briefly discuss the technological and process elements. We show how the model is a useful basis for comparing component generation technologies. With a rapidly growing area like component generation, it is hard to get a truly representative sample of generators. As a workaround, we illustrate our model using seven significant component generation systems developed by various research groups, and discuss some insights that the model provides. We conclude with an overview of the current status of our investigation.

1. The Pragmatic Need for Models

We know that component generation can be very beneficial for evolving systems, but we don't have a widely-accepted conceptual model for component generation systems. Conceptual models allow us to categorize and distill our knowledge of details into more manageable and structured information. We believe that such a model would facilitate better communication of ideas, within our own research group (PacSoft), within the component generation research area, within the programming languages area, and with the outside world. For example, it will necessarily play an important role in transferring our ideas as a research community to software houses that can develop industry-strength, general purpose component generators.

We have been working towards such a model for almost three years now, and have studied over 100 related publications, in addition to being involved in PacSoft's SDRR component generation project [KMB96]. Why has it taken so much effort? The major hurdle is that interesting component generation systems emerge from many corners of computer science, which often means incompatible vocabularies. For example, the word "Component" can have significantly different meanings in different papers[2]. The diversity of programming languages, operating systems, and tools used in developing the generators, and of the researchers' expectations from all of these, add significantly to the difficulty of understanding the literature in a manner that would allows us to compare and contrast two different generation technologies.

2. The Architectural Element

Software architectures [PW92] communicate ideas about software systems, and are especially useful when parties involved come from a variety of different backgrounds. Architectural descriptions provide an abstract basis for our model, a basis that is independent of the technology underlying the generator, the development process, and the application domain.

Even when composed of relatively simple subsystems, the collective architecture of a generator is often quite complex, and involves a significant number of distinct artifacts and users. Artifacts include the generator, the input and output of the generator, libraries, and the legacy system hosting the generated component. Users include the developers of the generator, it's input, and the libraries. Ideally, the input to the generator is a simple, compact specification that is easy to maintain. However, it is often the case that an executable program cannot be generated solely from such

[1] This research is supported by a contract with the USAF Materiel Command. Contract F19628-93-C-0069.

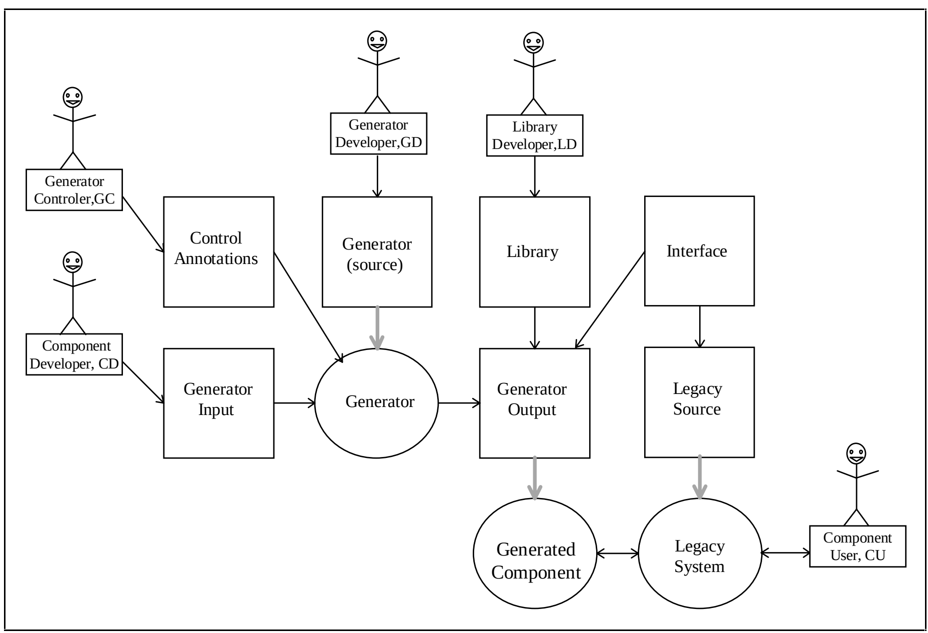
[2] In this paper, it will mean CORBA/COM-like components.

specifications. Therefore, it is common to find an additional (specification) language, often in the form of *annotations*, for controlling the generator. There may even be a developer dedicated to this task.

Hence, a model for component generators should admit all possible answers to the following questions:

- What is the input to the generator? Who writes this input?
- What is the output of the generator? Who uses it?
- What libraries does the output use? Who writes these libraries?
- How does the generator work? Who wrote it, and how? How is it controlled?
- With what systems does the generated component interact?

While it is not common to consider all of these dimensions of variability simultaneously, this is precisely what is needed when we wish to relate and contrast more than one existing component generation system. The figure below is a schematic[1] representing the minimal architectural schema that arises if the answer to the each of the above questions is distinct.



The figure above explicates the implicit complexity of even the simplest generative system. For instance, consider the yacc parser-generator [Joh75]. Development work on the generator itself has stopped, and hence, we usually don't think of either the developer or the source yacc.c. The generator input is the grammar proper, and the control annotations are the directives regarding precedence and association. Note that control annotations need not be in a separate file. The component developer and the generator controller are the same person. The grammar file could also contain further control instructions about what library files the generator output might be using. The libraries used by the generator output include lib.y.c, which contains the abstract machine for the parse table The interface is usually header files describing the legacy system functions that the parser uses. Finally, while we rarely see a user directly interacting[2] with the parser generated by yacc, the user of the legacy system is, indirectly, the component user.

[1] Drawn in the Generator Description Language, GDL [TS97].

[2] Interaction commutes, and hence, we could have drawn the component user directly connected to the generated component, and the diagram would have had the same meaning.

2.1 Basic Distinguishing Characteristics

Certain aspects of the architecture sketched in the last section are "not negotiable": a generative architecture has to include a generator, a generator input, and a generated component. And every artifact that is not mechanically generated must have an author. The architecture described above gives us a very natural basis for our model that captures these essential invariants. However, it offers too many dimensions of variability. The design space is indeed vast. But some of these dimensions are more informative than others, in that they are better discriminators between various component generation systems. We have identified *basic distinguishing characteristics*:

1. Who is the primary user, that is, the "customer" the system is intended to benefit?

2. What expertise is expected from the main user?

3. Which users are distinct, and which users are not? For example, is the role of generator development identified with the role of generator control?

4. Does the generator have a distinct notion of control annotations?

These factors are *derived* or *computed* from the architectural variabilities. In the following section, we illustrate the relevance of these criteria by considering some important generative systems.

2.2 Application to Seven Research Component Generation Systems

For brevity, we will not review all the systems we have studied. Instead, we present summary of our observations, and then illustrate how these observation can be interpreted. In the following table, "=" between two different kinds of users means that we did not find them to be treated differently. In cases where there is no explicit notion of control annotations, the input to the generator can be viewed as being an " *Implicit*" control specification:

Systems	**Primary User(s)**	**Primary User's Expertise**	**Distinct Users**	**Control Annotations**
ISI [Bal81,Bal92]	GD, CD	GD: Meta-programmer, CU: Domain expert	CU, CD=LD, GC=GD	Pragmas
MIP [MKS97]	CU	Domain expert	CU=CD=GC, GD, LD	*Implicit*
GenVoca [BST+94]	LD	Programmer	CU, CD=GC=LD, GD	Design rules
KIDS / SpecWare [Smi90, SJ94]	CD	Formal methods expert	CU, CD=GC=LD, GD	Refinements
SDRR [BH+94, KMB96]	GD, CD	Domain expert	CU=CD, GC=GD=LD	*Implicit*
Amphion [LPP+94]	CU	Domain expert	CU=CD, GC=GD, LD	*Implicit*
AOP [GLM+97]	CD	Programmer	CU, CD=GC=GD=LD	Aspects

Let us consider the first case: In the ISI technology, the generator developer (GD) uses the POPART meta-programming tool-kit and a relational extension of C or Java to develop the generator [Bal92,Wil81,Wil90]. In the literature we surveyed, the roles of the generator controller (GC) and generator developer were not distinguishable. Pragmas are used to guide the relational compiler as to how to implement relations.

The following sub-sections discuss two of the main observations that can be drawn on the basis of this information.

2.2.1 What to Mix, and What to Match

Consider the kind of information that might interest a software engineer interested in building a component generator. Some technologies address similar classes of users, such as ISI and SDRR, and MIP and Amphion. This means that these technologies could be a good basis for synthetic systems combining the benefits of both. For example, SDRR's technology, which leverages on functional programming, can benefit greatly from ISI's meta-programming technology, and vise versa. When a basic distinguishing characteristic identifies two systems, there are usually many other (often less-abstract) dimensions in which they are different. For example, MIP and Amphion fall on distinct points along the dimension of real-time constraints. We consider this dimension to be somewhat less abstract than architecture because it is more dependent on the application domain. Some of these dimensions should be in a model for component generators, discussed in the next section.

Other technologies address users that are usually not emphasized by others. For example, GenVoca is unique in addressing concerns of the library developer (LD). This suggests that high-level ideas from the GenVoca system might be readily combinable with generation technologies covered in our survey.

2.2.2 How to Control Generation

Four very different kinds of annotations are being considered by three different groups, namely, ISI's *pragmas*, GenVoca' *design-rules*, KIDS and SpecWare *refinements*, and AOP's *aspects*. These annotations are an important characteristic of modern component generation systems that was not commonplace in earlier transformational programming systems.

Control annotations can be viewed as Domain-Specific Languages (DSLs). For example, yacc's specifications for precedence of operators is one such DSL. In this light, we can say that the first three kinds of annotations are single languages, and AOP's aspects can be thought of as families of DSLs. We believe that the study of these generator-control DSLs will play an important role in developing general-purpose, industry-standard component generation systems.

3. Technology and Process Elements

Our model also includes two other elements; the technology underlying the generation system, and the process by which the generator itself is developed. Both can be viewed as refinements of the architectural model. The following table summarizes some distinguishing characteristics of the systems surveyed:

Systems	Underlying Technology	Generator Development
ISI	Meta-programming calculus and tools	Using POPART tools and relational C or Ada
MIP	Model-integrated real-time control	Using the MIP paradigm
GenVoca	Algorithm selection and object-orientation	Using design rules to specify acceptable library combinations
KIDS / SpecWare	Formal verification	Using specifications and refinements to characterize and derive programs
SDRR	Typed, functional programming	Using SDRR to create the front-end of the SDRR pipeline

Amphion	Theorem proving and program synthesis	Using Meta-Amphion, a theory of the domain, and an inference engine
AOP	AOP	Using (any technology?) to develop a weaver and aspects

4. Conclusion

We have outlined a model for component generation systems that we are currently developing. The model captures some of the bare essentials required for an object of study to be considered a generator, without going too deeply into the details of any particular system. We illustrated how it admits simple, clear, and objective criteria for comparing component generation systems. Our work shows that there is significant diversity not only in the cultures and application domains of contemporary component generation research projects, but also in technical problems that are unique to the emerging research area of component generation, such widespread interest in generation control.

***Acknowledgments**: We thank Lisa Walton, Andrew Black, Sherri Shulman, Tito Autry, Therese Fisher, Shailish Godbole, and Amol Vyas for valuable discussions.*

Refferences

[Bal81] R. Balzer, Transformational Implementation: an Example, *IEEE Transactions on Software Engineering*, Vol. SE-7, Number 1, January 1981.

[Bal92] R. Balzer, Design Refinement in DSSAs, *Proceedings of the JSGCC Software Initiative Strategy Workshop* , December 1992.

[BH+94] J. Bell, F. Bellegarde, J. Hook, et al., Software Design for Reliability and Reuse: A Proof-of-Concept Demonstration, In *TRI-Ada '94 proceedings*, pages 396-404, November, 1994.

[BST+94] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin, The GenVoca Model of Software-System Generators, *IEEE Software*, September 1994.

[GLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin, Aspect Oriented Programming, PARC Technical Report, February 97.

[Jon75] S. Johnson, YACC --- Yet Another Compiler-Compiler,*Computing Science Technical Report* , Bell Laboratories, No. 32, 1997.

[KMB96] R. Kieburtz, L. Mckinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith, L. Walton, A Software Engineering Experiment in Software Component Generation, *Proceedings of 18th International Conference on Software Engineering*, Berlin, IEEE Computer Society Press, March, 1996.

[LB95] M. Lowry, J. Van Baalen, Meta-Amphion: Synthesis of Efficient Domain-Specific Program Synthesis Systems, *Proc. of 10th Knowledge-Based Software Engineering Conference* , Boston, Mass, Nov. 12-15, 1995, pp. 2-10.

[LPP+94] M. Lowry, A. Philpot, T. Pressburger, I. Underwood, Amphion: Automatic Programming for Scientific Subroutine Libraries, in *Proc. 8th Intl. Symp. on Methodologies for Intelligent Systems* , Charlotte, North Carolina, Oct. 16-19, 1994, pp. 326-335.

[MKS97] A. Misra, G. Karsai, J. Sztipanovits, A. Ledeczi, M. Moore, E. Long, A Model-Integrated Information System for Increasing Throughput in Discrete Manufacturing, *Proceeding of the Engineering of Computer Based Systems (ECBS) Conference*, Monterey, CA, March 1997.

[PW92] D. Perry and A. Wolf, Foundations for the Study of Software Architecture, *ACM SIGSOFT Software Engineering Notes*, 17:4, October 1992.

[SB97] Y. Smaragdakis and D. Batory, DiSTiL: A Transformation Library for Data Structures, *USENIX Conference on Domain-Specific Languages*, October 1997.

[SJ94] Y. Srinivas and R. Jullig, SpecWare: Formal Support for Composing Software, *Proceedings of the Conference on Mathematics of Program Construction*, Kloster Irsee, Germany, July 1995. Kestrel Institute Technical Report KES.U.94.5.

[Smi90] D. Smith, KIDS: A Semi-Automatic Program Development System, *IEEE Transactions on Software Engineering* --- Special Issue on Formal Methods, Vol. 16, No. 9., September 1990.

[TS97] W. Taha and T. Sheard, Facets of Multi-Stage Computation in Software Architectures, *OGI Technical Report CSE-97-010*, September 1997, Oregon Graduate Institute.

[Wil81] D. Wile, POPART: Producer of Parsers and Related Tools. System Builders' Manual, Technical Report, USC Information Sciences Institute, 1981.

[Wil90] D. Wile, Adding Relational Abstraction to Programming Languages, *ACM SIGSOFT Software Engineering Notes*, 15(4), pp. 128-139, September 1990.