Encyclopedia of Electrical and Electronics Engineering, (John Wiley and Sons).

Application Generators

Yannis Smaragdakis and Don Batory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712

1 Introduction

When a programming activity is well-understood, it can be automated. Automation transforms software development from activities like rote coding and tedious debugging to that of specification, where the "what" of an application is declared and the "how" is left to a complex, but automatable mapping. Programs that perform such mappings are *application generators* (or just *generators*). In the technical sense, application generators are compilers for *domain-specific programming languages (DSLs)*. There is no strict criterion for characterizing a language as "domain-specific" but the term is commonly used to describe programming languages for specialized tasks (as opposed to "general-purpose" programming languages). Examples are languages for implementing communication protocols, partial differential equation solvers, windowing software, etc. Although all compilers can be viewed as generators, generator research and practice has focused on problems different than those usually found in a classical treatment of compilers (e.g., [1]), such as programming language extensibility and program transformations.

Before we delve further into generator specifics, it is worth addressing the following question: why are generators needed? Is it not sufficient to employ other programming tools (e.g., traditional software libraries)? One answer is that it is very hard to scale traditional tools to handle code that is highly complex, yet can be decomposed into simpler pieces in systematic ways. In this case, generators can be viewed as compact representations of software libraries of gigantic size—each library encoding all the useful code configurations that a generator can produce. With a generator, the same code is produced in a systematic way that allows a mechanical process to generate desired code configurations. Thus, for practical reasons, generators are the preferred way to represent large families of related applications. Another reason for using generators is that the specification languages that generators implement are much more concise and convenient than the language of the produced program (called the *target language*). The translation of specifications to target code is done correctly and quickly, thereby substantially increasing programmer productivity. Further, generators can apply domain-specific optimizations (which are tedious and error-prone if done by hand) and can perform advanced error-checking (thereby automating correctness checks performed by domain experts). Compared to traditional software libraries, generators offer a much greater potential for optimization and can provide better error-checking.

There are many dimensions of variability among generators, despite their common goal. Some generators implement specification languages that have a sound theoretical basis (e.g., [44][48]) and thus have been used extensively to implement formal specifications. More typically, full axiomatic theories simply do not exist and generator design is based on an informal understanding of a domain.

Another important variability deals with implementation technology. Most generators are self-sufficient, stand-alone translators (in much the same way as compilers for general-purpose languages). Yet others take on a very different form, such as when generators are implemented using program transformation systems (e.g., [29][37]). A *program transformation system* (or just *transformation system*) is a platform for expressing and executing *program transformations*—that is, mappings from programs to programs. Sets of transformations define the automatable mappings of a particular domain. In this case, a generator would be merely a set of transformations that may not even be encapsulated in a single module.

In general, the field of application generators is a collage of ideas from various areas of computer science, such as programming languages, compiler technology, and software engineering, to name a few. An examination of the field reveals a few common principles and many distinct generator "camps", each promoting a different philosophy of what generators represent and how they should be built. Hence, a representative overview of generators needs to include both basic background and a sampling of approaches. This article cannot be fully comprehensive, however. For further reading, our references emphasize recent work which can serve as a starting point. Partsch and Steinbrueggen [34] provide a good survey of past work on transformation-based systems.

2 Architecture of a Generator

Application generators have the standard internal form of a compiler with a front-end, translation engine, and back-end component (see Figure 1). The front-end is responsible for the one-to-one mapping of the input form to an equivalent but more convenient internal representation, such as flow graphs, or abstract syntax trees, possibly annotated with data-flow and control-flow information. Typical input specifications are in text format, in which case the front-end consists of a conventional lexical analyzer and a parser. Other specification formats (e.g., graphical representations) may map straightforwardly to the intermediate representation, thus simplifying the front-end. It is interesting to note that generator writers have often tried to keep the cost of implementing front-ends low by employing ideas from extensible programming languages. Many generators are implemented as extensions of the Lisp language [49] or its variants. Lisp has explicit syntax (mapping directly to parse trees) and a very powerful extension facility (Lisp macros). In other cases, tools that generate parsers from modular grammar specifications (e.g., [6][36][51]) have been used. Such tools can effectively extend a language by adding new language constructs.

The translation engine implements transformations on the intermediate representation. Usually transformations are expected to satisfy some correctness property: the transformed program should have the same semantics as the original, if not for all inputs, at least under well-defined input conditions. Translation engines and transformations are the core of generators and are discussed in detail in the next section.

The result of applying transformations to the intermediate representation is a concrete executable program. The concrete program, however, is still represented as a flow graph or an abstract syntax tree. Mapping from the intermediate representation to program text is straightforward and is the role of a generator's back-end. Generated code is usually in a high-level programming language. Several generators and transformation systems (e.g., [2][20][29][48]) offer multiple back-ends, thus producing code in more than one language. Once again, this can be a straightforward process, if the generator does not rely on unique features of any specific language.

3 Transformations in Generators

Translation engines and the transformations they support are the heart of all generators. In the next two sections, we classify the most common kinds of transformations with respect to two criteria. First, Section 3.1 describes transformations from a technical standpoint. This answers the question of *how* translation engines express and apply transformations. Second, Section 3.2 discusses *what* transformations accomplish.

3.1 Transformation Machinery

There are several degrees of variability in the capabilities of translation engines (and, consequently, in the transformations they support). A fairly comprehensive classification of translation engines can be derived by answering the following questions:

- How are transformations expressed? (E.g., procedurally, using syntactic patterns, or using data-flow patterns.)
- How powerful are they? (E.g., can they change the global outline of a program or only local properties?)
- When are they applicable? (E.g., can they depend on complex data-flow properties? How are such properties expressed? What is the machinery to check them?)
- If multiple transformations are applicable, what is the order of their application? (E.g., are transformations always applied in a fixed order? If not, is the order determined automatically or manually?)

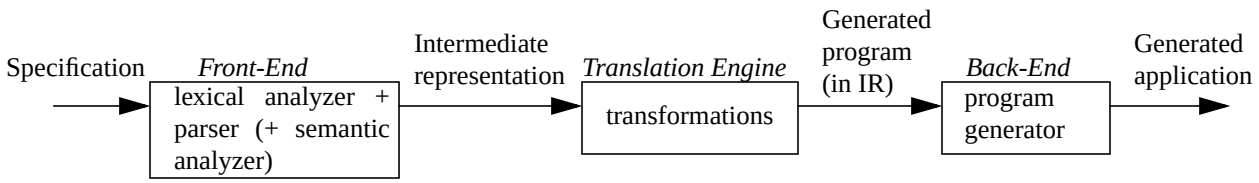

Figure 1: A generator is similar to a conventional compiler, with a front-end, translation engine, and back-end.

- To which extent are transformations automated? (E.g., does the user need to explicitly match program elements to transformation elements, or is the matching automatic?)
- Is the set of transformations fixed or extensible? (E.g., can the user add new transformations? Can the translation engine combine existing transformations to form new ones?)

Instead of answering each question individually, we identify four common axes of variation in transformation engines:

1. **Stand-alone generators vs. general transformation systems**: Generators can be packaged either as stand-alone tools, in much the same way as regular compilers, or as collections of transformations under a general transformation system. Expressing a generator as a collection of transformations has the disadvantage of making the generator dependent on a complicated piece of infrastructure (the transformation system). On the other hand, transformation systems (e.g., [8][28][37]) offer support for expressing and applying a variety of transformations in a general, domain-independent way. In other words, both the language in which the transformation is expressed, as well as the mechanism that applies transformations are determined by the transformation system and are the same for every domain. In theory, generators expressed as a collection of transformations are easily extensible, simply by adding more transformations. (In practice, this generally is not the case. There are often subtle interdependencies among individual transformations that make transformation additions and substitutions hard and error-prone.) Additionally, because transformation systems are domain-independent, they typically allow for a higher degree of sophistication in the translation engine. Thus, general transformation systems commonly support specifying transformations declaratively instead of operationally. Given the declarative specifications of transformations, the translation engine may be able to deduce the appropriate order of transformation application when multiple transformations are applicable. Also, optimizations in the transformation application may be possible, by combining transformations to form new ones.
2. **Programmatic vs. pattern-based transformations**: As mentioned earlier, a transformation is a mapping from an input program to an output program. Such mappings can be expressed in a variety of ways. Transformations are commonly classified as *programmatic* or *pattern-based*. Programmatic transformations are arbitrary programs that manipulate code representations (also known as *meta-programs*). Pattern-based transformations, on the other hand, are written in a special pattern language that repeatedly searches a program representation for instances of patterns. When a pattern is found, the transformation is applicable and may be triggered, resulting in a different pattern being replaced for the original one. (Such a transformation is also known as a *rewrite rule*.) For instance, a simple rewrite rule could have the form:

   ```
   while (cond) stmt  ->  L: if (cond) { stmt; goto L; }
   ```
   (1)

   The left hand side of the above rule is the pattern to be matched, and the right hand side is the pattern to be replaced. (Patterns are written in self-explanatory syntax resembling that of the C language.)

   Both pattern-based and programmatic transformations offer distinct benefits. Pattern-based transformations are generally simple and easy to understand. At the same time, their declarative nature allows for sophisticated automatic manipulation. In theory, pattern-based transformations are as expressive as any computer program (equivalent to Markov systems, e.g., see [23], p.263-264). Practically, however, pattern-based languages are inconvenient for applying complex transformations that rely on complex properties or contextual information. Programmatic transformations overcome this restriction. Overall, programmatic transformations are usually employed in *ad hoc* generator systems (i.e., stand-alone compilers for a specific domain) or for expressing global program transformations. Pattern-based transformations are in wide use in general program transformation systems. It is also possible to mix pattern-based and programmatic transformations. For instance, a transformation may be triggered by a certain pattern but the actions executed at this point may be specified programmatically. Similarly, a transformation may be programmatic but use patterns to describe newly created code.

   Many interesting languages for expressing transformations are hard to characterize as strictly pattern-based or programmatic. For instance, a large number of transformation systems (e.g., [19][38]) rely on attribute grammars [21] for expressing transformations. Briefly, attribute grammars are context free grammars extended with syntax-directed functional (i.e., side-effect-free) computations of “attribute” values, which are associated with symbols in the grammar. Thus, transformations expressed as rules in attribute grammars are triggered by parsing (essentially, pattern-matching) the program representation. Nevertheless, the actual action performed when a rule matches is expressed in a limited programmatic form. Limiting the attribute computation to be functional allows the translation engine to determine automatically the order of transformation applications, based on the dependencies among attributes.

3. **Syntax-directed vs. flow-directed transformations**: As in standard compiler-based transformations, the translation engine of a generator could be operating on intermediate representations that reflect syntax (e.g., abstract syntax trees) or control/data flow (e.g., flow graphs). Syntax-based representations have the advantage of being simpler, easier to obtain, and directly reflecting the hierarchical nature of the program to be transformed (e.g., a `while`-statement is represented as a tree with the `while` operator at its root). Control flow-based representations have the advantage of providing a normal form for representing control information (e.g., all kinds of loops have the same form in a flow graph).

   The vast majority of realistic transformations are only applicable under certain guarantees about the *context* of a transformation application site. For instance, the following two transformations are context-dependent:

   x evaluates to a number, has no side-effects `=> (`*x*` + 0 -> `*x*`)` (2)

   cond is guaranteed true, has no side-effects `=> (if (`*cond*`) `*thenbody*` else `*elsebody*` -> `*thenbody*`)` (3)

   The transformations are read as follows: if the current context implies the property on the left side of the "=>" symbol (called the *enabling condition*), then the rewrite rule on the right side of "=>" is applicable. To enable context-dependent transformations to be applied automatically, the generator must perform extensive program analysis. This analysis is easier with a program representation that makes the program's control and data flow explicit (e.g., a flow-graph). (A discussion of program analysis techniques is beyond the scope of this article, and can be found in textbooks on optimizing compilers—e.g., [26].)

   In practice, unlike general-purpose compilers, few generators use intermediate forms that explicitly reflect control flow. Notable exceptions are stand-alone generators for domains that are best exploited by traditional compiler analysis tools (e.g., matrix algebra [24]). Only few general transformation systems (e.g., [8]) use a control flow-based representation, but almost all support the annotation of abstract syntax with information derived from program analysis. The motivation behind this widespread practice is partly its simplicity, but also the fact that generators usually transform *generated programs* and not arbitrary programs that an end-user has written. That is, control/data flow analysis is rarely meaningful at the level of the input of a generator. Most generators have input languages that are highly declarative, with very little operational information. When a generator transforms the input specification, it can produce at each step both the transformed code and automatically derived *properties* of this code, which can be attached as annotations (e.g., see [11]). In this way, one transformation step can supply all necessary contextual information to the steps following it, thus avoiding the need for program analysis. For this approach to be successful, the generator writer has to identify in advance a few high-level properties that are fundamental for the produced implementation (e.g., the property "expression *e* has no side effects" for transformations (2) and (3), above).

   Based on the above observation, it is not surprising that the emphasis in generators (beyond program synthesis) is not on program analysis (deriving program properties) but on expressing program properties and inferring other properties from them. Thus, generators and transformation systems often offer powerful inference capabilities, in the form of specialized theorem-provers (e.g., [43]).

4. **Degree of structure in the transformation process**: The spectrum of translation engines found in generators is very wide. A good heuristic rule for classifying generators is to compute the average number of transformations that are potentially applicable at every step in the transformation process (i.e., how many options the system has when it makes a transformation decision). For stand-alone generators, whose input is a rather concrete specification (e.g., [5][16][24][42]), this number is typically small (at most around 10). Furthermore, the transformation process in simple generators may be *confluent*: different orders of transformation application can produce different intermediate results but further transformation will reduce them all into the same normal form. More ambitious generators, translating more abstract specifications (e.g., [12][44]) usually have to choose among many tens or hundreds of transformations at every step. In other words, generators of the first kind act more like conventional compilers, while generators of the second kind apply more intelligence in the transformation process, using heuristic knowledge to make complex decisions.

   Some of the latter generators (e.g., [44]) are based on an *equational rewrite* paradigm. That is, transformations may be specified only implicitly using a set of axioms in an equational logic. The generator can then use these axioms to derive equational properties (theorems). Each of these equations can be viewed as a pair of transformations: either the left hand side can be matched and the right hand side be replaced, or the converse. In this case, it is not easy to guarantee that the transformation process will always terminate. A naive transformation engine may even repeatedly perform a transformation and its reverse, as they are both derived from the same equation. There has been significant work on deriving (from a set of equality axioms) a set of transformations that are guaranteed to terminate, regardless of their

application order. Most work is based on the well-known Knuth-Bendix completion algorithm [22] and a relatively recent comprehensive survey of rewrite systems can be found in [14].

Although a sophisticated transformation process is desirable, it can also be highly complicated. "Traditional" transformations are rewrite rules that work on a small fragment of code, such as (1)-(3) above. Given a set of such rules, automatically determining the next rule(s) to apply may be very difficult, and hence it is not uncommon for transformation systems with such rules to require periodic guidance/inputs from its users. The degree of interaction becomes more involved as programs become more complicated: Transforming a declarative specification into an optimized program may require many thousands of such rewrites. To address this complexity, many modern generators (e.g., [5][16][42]) encapsulate several small transformations in large components and apply them in a consistent manner (i.e., the generator decides to apply either all the transformations in a component or none). This approach, known as "consistent refinement", is quite beneficial in the domains for which it is applicable (typically such domains are well-structured and well-understood). For example, suppose one is transforming a declarative specification of a program that uses a data structure. At one point in the translation, a concrete implementation must be chosen for the data structure. A large number of small transformations may make a common assumption (e.g., the data structure is a list), and all of them need to be applied consistently.

3.2 Common Kinds of Transformations

Transformations can usually be classified as refinements or optimizations. A *refinement* adds implementation detail to an abstract specification. For instance, an *abstract data type*, like a set, may be refined to be implemented using a specific *data structure*, like a binary tree (e.g., [12]). Refinements can occur at many levels and may fundamentally affect program structure and performance. *Optimization* transformations attempt to improve the performance of the produced program by transforming it into a more efficient program, at the same conceptual level of abstraction.

Generators are fundamentally about refinements (i.e., taking abstract specifications and progressively making them concrete). Nevertheless, compared to compilers for general-purpose languages, generators offer more opportunities for optimization transformations: code that is automatically produced from a high-level specification often exhibits a greater potential for optimization because generated code is usually highly structured. Domain-specific knowledge (which the generator incorporates) can be used to exploit this structure to realize fairly sophisticated optimizations. In all, the difference in performance between optimized applications and unoptimized ones may well be up to several orders of magnitude.

Next we discuss some common kinds of refinement and optimization transformations. Our presentation is selective. A valuable further reference is Partsch's textbook [35], which contains a large number of example transformations for many common refinement and optimization tasks.

3.2.1 Refinement Transformations

The presence of refinement transformations is the single most striking difference between generators and compilers for general-purpose languages. We discuss two common types of refinements below:

1. **Algorithm Derivation**: The most important kind of refinement for generators is that of transforming a declarative specification into an operational procedure that produces values satisfying the specification. Common algorithm derivation transformations include mapping operators from the declarative specification into heuristic-guided search procedures. For instance, an existential quantification (i.e., a specification of the form "there exists an element satisfying property *P*") can be mapped into a search procedure that iterates over elements until one is found to satisfy property *P*. The challenge is to exploit the structure of property *P* and use it to derive efficient implementations that do not exhaustively search the space of possible solutions. For instance, *P* could be a property that admits efficient filtering (i.e., if there is an element satisfying it, then a larger group of elements will satisfy another property *Q*, which can be used to filter out non-solutions). Excellent starting points for exploring the wealth of research work in general algorithm derivation are Chapter 5 of [35], and [44][46].

   Deriving algorithms from highly abstract specifications is still a research challenge, however. In practice, most actual generator systems are less ambitious. Stand-alone generators (e.g., [5][11][17][24][31][42]) usually perform algorithmic refinement by using *algorithm schemas*: generic algorithm templates that allow limited specialization for particular data representations and special-purpose operations. For instance, an algorithm schema could provide the skeleton of a global search procedure. This procedure can then be specialized by adding the actual conditions for terminating the search. Local optimizations can be performed, but the overall structure of the search process will be the same for every

search procedure generated, regardless of data structure or searched element. Clearly, this approach can only produce efficient code for highly structured domains, but this is sufficient for most generators that cater only to specific programming needs.

2. **Data Type Refinement**: A complementary refinement to algorithm derivation is that of selecting an implementation for data types in a specification. Different data structures offer good performance for different operations (e.g., retrieval of elements with key values in a range, vs. retrieval of elements with a single key value). Additionally, often data structures need to be combined, effectively creating *indexes* that support the efficient retrieval of groups of elements. Just like in the case of algorithm derivation, the approaches taken by different systems vary with respect to their sophistication. Systems that take input in a declarative language often use a set-theoretic abstraction for specifications. Sets can later be mapped into efficient data structures automatically (see Chapter 9 of [35], and [39][44]). The choice of data structure depends on the kind of operations commonly performed (e.g., exhaustive searches vs. searches that can be efficiently indexed). At the same time, the guarantees offered by the data structure (e.g., always fully sorted vs. partial priority queue ordering) influence the way algorithms are derived. For instance, the decision to choose a fully sorted data structure may influence the subsequent choice of an algorithm that manipulates data structure elements. The interplay of algorithm derivation and data type refinement provides interesting research challenges.

   Many generators (e.g., [30][42]) employ less ambitious approaches to data type refinement, by allowing the user to specify either the desired data structure, or the desired algorithms, and optimizing one choice based on the other. An example of this approach is discussed in Section 4.2.

3.2.2 Optimization Transformations

Optimization transformations in generators partly borrow from conventional compiler technology. Nevertheless, several kinds of optimizations have been developed much more extensively in the transformational programming community than in general-purpose compilers. Optimization transformations in generators fall mainly into three categories:

1. **Partial Evaluation**: *Partial evaluation* (e.g., see [10]) refers to the specialization of a code fragment under the assumption that its (implicit or explicit) parameters satisfy certain conditions. It is probably the most common kind of optimization in application generators (for instance, transformations (2), and (3), shown earlier, represent cases of partial evaluation). This is expected: partial evaluation is a general technique for specializing general pieces of code for use in concrete contexts. Partial evaluation can be effected through pattern-based transformations but the most complex cases are usually treated programmatically. Two special cases of partial evaluation are *specialization* (producing a new function by fixing some of the arguments of an existing one) and *constant folding* (performing computations on constants at compile time).

2. **Incrementality Optimizations**: Another class of valuable optimizations rely on techniques that perform complex computations incrementally. This is particularly interesting in the context of generators since, when composing abstract algorithms, a generator often has knowledge of the update patterns for the data used by each algorithm. Thus, it is not surprising that incrementality optimization techniques have been explored extensively in the generator community. One such technique is known by the name *finite differencing* or *formal differentiation* [32][33][40]. Finite differencing substitutes expensive computations that occur in a specific pattern (e.g., in a loop) with an incremental update of the result of the previous computation in the pattern. The origins of finite differencing can be traced in the well-known *strength reduction* optimization in compilers. Continuing work in transformational programming has yielded new results in a more general setting (a good starting point for exploring recent research is [25]).

   Finite differencing is best applicable when there are strong static guarantees on how data are updated. Other incrementality optimizations can be used even when a strong pattern is not statically known, but run-time uniformities are expected. This is the case with the *caching* or *memoization* optimizations (e.g., see Chapter 6 of [35]). These techniques store values produced by a computation at run-time so that they can be used by subsequent operations (possibly for incrementally computing other values). The algorithms used need to be modified to take advantage of cached values when these are available.

3. **Code Restructuring**: Several control- and data-flow based optimization techniques fall under the general heading of *code restructuring* transformations. These include most traditional compiler optimizations like *dead code elimination*, *loop unrolling*, *loop invariant code motion*, as well as techniques like *loop fusion* (see [26]). The applicability of such optimizations can either be inferred from the code or established by previous refinements, so that expensive program analysis infrastructure is not required.

4 Case Studies of Contrasting Approaches

As indicated earlier, there is significant variability among generators: generators are being used for everything from trivially automatable specifications to formal languages that cannot be transformed without human input. Additionally, generators are built using widely different techniques. In this section, we look at the approaches taken in two generators that are, in many respects, at opposite ends of the spectrum. (Many more (older) systems are discussed in [34].) Each of the two generators that we have selected are among the best-known representatives of a distinct and wide class of successful systems. At the same time, each promotes a distinct philosophy on the principles upon which generators should be based. We end this section with a comparison of these approaches.

4.1 KIDS

The *Kestrel Interactive Development System (KIDS)* [44] is a semi-automatic generator applied to the problems of *automatic programming*. Although it is hard to strictly define what "automatic programming" is, the name is usually reserved for the most ambitious software production techniques, i.e., those trying to automate most of the software development process. Even though automatic programming has been a moving target (the first compilers were touted as "automatic programming" systems), a consensus on the fundamental elements of the field has evolved in the past three decades (sadly reflecting our failure to advance the "automation" target significantly during this period). Two main approaches to automatic programming are usually identified: the knowledge-based approach and the formal-model-based approach. KIDS is one of the primary representatives of the formal-method-based approach. More importantly, in addition to its ambitious goals, KIDS has seen several practical applications and has tested the limits of common generator optimizations.

The domain of KIDS is that of algorithm design and implementation. The system superficially departs from the usual generator model since several high-level transformation decisions are specified interactively by the user. Nevertheless, it is fundamentally a generator that refines and optimizes a formal specification. The input of KIDS is a functional specification of a problem (i.e., a function characterizing the possible outputs for each input) expressed using first-order logic operators and set-theoretic data types. As a simple example, the notion of an injective sequence of integers (sequences can be viewed as functions with a domain *1...n*) can be expressed as:

`function` *injective(M: seq(integer), S: set(integer)): boolean*

$$= \; range(M) \subseteq S \wedge \forall(i, j)((i \in domain(M)) \wedge j \in domain(M)) \Rightarrow (i \neq j \Rightarrow M(i) \neq M(j))$$

That is, a sequence *M* is injective into a set *S* if all elements of *M* are in *S* and no two elements of *M* are the same. Distributive laws are common in KIDS specifications, essentially specifying a structural induction phase: the meaning of the combination of two operators is defined in terms of the meanings of "simpler" combinations. An example distributive law for the *injective* predicate is:

$$\forall(W, a, S)(injective(append(W, a), S)) \; = \; (injective(W, S) \wedge a \in S \wedge a \notin range(W))$$

KIDS gets additional input interactively from a human user. The user can make strategic decisions like "design a divide-and-conquer algorithm for this specification" or "simplify this algorithm by applying finite differencing on this value". The system contains a powerful inference engine [43] that applies pattern-based transformations derived from theorems of first-order logic. To schedule these transformations, the engine uses a combination of heuristic measures, such as the number of logical "weakening" rules that it has applied. KIDS encodes knowledge of a few general algorithmic search procedures (like "global search") in the form of program templates. The result of the inference procedure is a correct specialization of such templates, thus yielding a complete abstract algorithm.

At that point, standard refinement and optimization techniques can be applied to the output. KIDS provides several rewrite rules for either context-independent (i.e., without enabling conditions) or context dependent simplifications. The powerful inference infrastructure collects context information and decides whether an expression can be simplified. Other optimizations (different forms of partial evaluation and finite differencing) can also be applied under user guidance. Finite differencing, in particular, is especially valuable because of the set-theoretic nature of KIDS specifications. Sets can easily be specified incrementally and most KIDS algorithms reference complex predicates on sets. Refinements are also essential in KIDS to implement abstract data types (such as sets, maps, and sequences) as efficient data structures (e.g., arrays, trees, and lists).

KIDS is a representative of a formal approach to the specification of a domain. Assessing its applicability is hard—there is no general algorithm for satisfying specifications in first-order logic. Thus, we can only judge the practical value of the

KIDS approach in empirical terms. In these terms, the system has been successful. Its best known application has been in deriving very fast and accurate transportation schedulers for use by the U.S. Transportation Command [47]. Excellent discussions on the application of KIDS to other (simpler and more easily understood) domains, together with complete examples of program derivations, can be found in [44] and [45].

Many other generator and transformation systems efforts are directly related to KIDS. The system is built on top of the Refine [37] transformation system (currently marketed under the name *Reasoning5*). In fact, the input specification language of KIDS (logic-based with set-theoretic types) is part of the standard Refine infrastructure. Refine also offers a front-end tool [36] for the creation of modular parsers and a back-end (unparser) tool. Internally, programs are represented as abstract syntax trees, data-flow graphs, or control-flow graphs, depending on the most convenient level for each manipulation. Finally, many of the ideas introduced in KIDS relative to specifying search theories formally are more systematically explored in the SPECWARE system [48]. SPECWARE is mainly concerned with modeling domains using algebraic specifications and composing specifications using techniques motivated by category theory.

4.2 P2

P2 is a *component-based* generator for the domain of container data structures. Component-based generators (e.g., [13][16][17][27][42]) are a common class of generators whose transformations are represented as reusable and interchangeable components. Users declaratively specify their target application (in this case, a container data structure) and use compositions of components to tell the generator how to transform these declarations into efficient code. By using different compositions of components, P2 generates a completely different implementation of the same declarative specification. The key distinction between a P2 component and a KIDS transformation is one of scale: a P2 component encapsulates complex refinements and optimizations of multiple data types and operations on these types, which are presented as a "monolithic" transformation to a user. As each P2 component has a simple interpretation (e.g., there are different components for red-black trees, ordered lists, etc.) and the P2 component library is quite small (a characteristic of all component-based generators), the number of components (or transformations) that have to be composed to specify even complex applications (e.g., data structures) is modest (~5-15).

P2 imposes relational abstractions on container data structures: data structures implement *containers* of elements and individual elements are accessible through *cursors*. Common data structures—arrays, binary trees, ordered lists—implement the container abstraction and are encapsulated as individual P2 components. P2 components implement protocols by which a component can query other components about what properties they support, what optimizations they can perform, what is the expected complexity of the code they generate, which other components they are compatible with, etc. Such knowledge is needed when generating efficient application source code, as well as when checking the consistency of component compositions.

The P2 language is a superset of the C language, where C is extended with cursor and container declarations and operations on their instances. For example, consider a phone-book data structure and the following declarations:

```
Container <phonebook_record> phonebook;              /* abbreviated container decl. */
Cursor <phonebook> where "$.phone == 4783487" joe;   /* cursor declaration */
Cursor <phonebook> where "$.name > "S" && $.name < "T"" all_s;/* cursor declaration */
```

Assuming that elements are instances of the `phonebook_record` record type, the first line above declares a container (`phonebook`) for such elements. (Actually, the container declaration is abbreviated from the usual P2 syntax since it does not specify the components that implement the data structure—see below.) The following lines declare two different cursors ranging over selected elements of the `phonebook` container. For example, the `joe` cursor ranges over all elements of `phonebook` where the phone attribute equals `4783487`. Predicates need not be this simple; P2 can handle arbitrarily complicated predicates. In addition, P2 offers the standard operations on containers and cursors. For instance, the `foreach` operation is used below to iterate over all elements accessible by cursor `all_s`, and for those selected elements, the name of the element is printed:

```
foreach( all_s ) { printf( "%s\n", all_s.name ); }
```

Container implementation decisions are controlled by the P2 user by composing components from the P2 library. This is achieved with a `typeq` (type equation) declaration:

```
typeq { simple_typeq = top2ds[qualify[hash[phone,odlist[name,malloc[transient]]]]]; }
```

`simple_typeq` is a composition of six P2 components, where each component encapsulates a consistent data and operation refinement of the cursor-container abstraction and is responsible for generating the code for this refinement. The `top2ds` layer, for example, translates `foreach` statements into `while` loops and primitive cursor operations; `qualify` translates qualified retrieval operations into `if` tests and unqualified retrieval operations; `hash` stores all elements in a hash structure where attribute `phone` is hashed; `odlist` connects all elements of a container onto a doubly-linked list that is ordered on ascending `name` values; `malloc` allocates space for elements from a heap; and `transient` allocates heap space from transient memory. The complete container declaration for the `phonebook` container is shown below; it declares the type equation that determines how the container is to be implemented.

```
Container <phonebook_record> using simple_typeq phonebook;
```

The type equation determines how elements are to be stored and which fields are to be indexed (e.g., attribute `phone` is hashed and elements are arranged on a list in ascending `name` order). The P2 generator is responsible for implementing all operations on cursors and containers efficiently using information that it can infer statically from cursor selection predicates and the container type equation. For instance, P2 infers for the `joe` cursor (above) that the fastest way to find elements that satisfy the predicate is to use the hash table on the `phone` field. Similarly, P2 infers for the `all_s` cursor that the fastest way to find elements that satisfy the `all_s` predicate is to traverse the `name`-ordered list. The techniques that P2 uses to evaluate the cost of each retrieval method are motivated by query optimization in database systems.

In essence, type equations relieve P2 of the burden of making high-level refinement decisions. P2 does not attempt to automate data structure (or type equation) selection, but rather offers a friendlier interface to the user and facilitates program modification when requirements change. This was demonstrated, for example, when P2 was used to re-engineer a hand-coded, highly-tuned container data structures used in a production system compiler (LEAPS). As a result, P2 reduced the code size by a factor of three and offered significant performance benefits (up to several orders of magnitude in some cases) [5].

It is worth noting that a more recent experimental P2-like generator, called P3, does support fully-automatic selection and optimization of type equations [7]. P3, like P2, represents the design of a data structure as an equation (i.e., type equation). Knowledge of when and how particular components can be used is expressible as an equation rewrite. By supplying a usage profile that defines the major operations to be performed on the data structure and their execution frequency, standard rule-based optimization techniques can be used to optimize type equations. Thus, by declaratively specifying how a data structure is to be used, P3 tools can derive automatically the sequence of transforms/components to compose to generate an efficient implementation. The reason why this is possible is that the combination space for composing P3 “large-scale” transformations is significantly smaller than the space of composing “microscopic” transformations.

P2 covers a well-known domain and, hence, is ideal for demonstrating the benefits of component-based generators over traditional software libraries. P2 components capture features that are not easy to compose in their concrete form. Components like a hash table and a linked list data structure will have very different interfaces if encoded as concrete library components. This is, for instance, true in the C++ Standard Template Library (STL) [50] where sequences and associative containers have different interfaces (and, thus, are not interchangeable). In contrast, P2 raises the level of abstraction up to the point where all data structures have the same interface. At the same time, the specification language (i.e., the selection predicates, discussed previously) supplies enough information to the generator so that the full functionality of individual components (e.g., the fast random access capabilities of a hash table) can be utilized. This way implementation efficiency is regained automatically by the generator, even though an abstract language is used for operation specification.

4.3 Comparison

Consider KIDS and P2 both from a technical and from an end-user perspective.

- KIDS is built on top of a general transformation system (Refine), whereas P2 is a stand-alone compiler.
- KIDS is a semi-automatic development system that takes general declarative specifications (first-order logic formulas) as input. P2 is highly specialized for a single domain (data structure programming), and its input contains significant (albeit compact) implementation guidance in the form of a type equation.
- KIDS is based on an equational rewrite engine and uses a complex inference engine to guide the transformation process. P2 has a straightforward translation engine, based on a combination of programmatic transformations and pattern-based macro expansion. In a typical transformation step, KIDS has a wide space of possible choices for the next transformation, whereas P2 has no more than a handful.

- Context information in KIDS is expressed in a rich language and can be combined to derive new properties. P2 only uses a small set of predefined context properties that guide the transformation process.
- KIDS has a sophisticated model for deriving new algorithms, while P2 can only specialize existing algorithm templates. Accordingly, the KIDS refinement process may require significant user interaction, while P2 is fully automatic.

The sharp contrast between our two generator case studies illustrates the heterogeneity of the area, despite the occasional technological similarities. Generators vary as much as the different domains of software, both in depth and in breadth. Generator technology can be quite practical and immediately applicable, as long as the domain of the generator is narrow, well-structured, and well-understood. At the same time, generator technology can be ambitious, tackling domains that have little structure and challenge the limits of our capabilities.

5 Application Generators Now and in the Future

5.1 Generators in Practice

Application generators represent a significant software production technology. The breadth of the application generators field allows it to claim successes in many practical settings. Bassett's *frames* [4] are a generative technique for adapting code text through pure lexical manipulation. Despite their simplicity, frames have been used with great success to create programs of significant size (e.g., million-line) in the information systems domain. Also, many programs that produce code skeletons by composing code templates are primitive generators (e.g., the *wizards* supplied with Microsoft compilers). Similarly many language tools for mature domains are clearly generators (for instance, the *yacc* parser generator or the *LaTeX* set of typesetting macros). Nevertheless, these are rarely considered examples of what we will call the *generator approach* to software development. The reason is that the above tools do not need sophisticated transformation machinery. For instance, typically such tools do not have to choose which transformations to apply, either because their domain is so well-structured, or because their job is simply to concatenate code text. Hence, the approaches that we examine here are among those that employ transformation technology of the kind discussed in this article.

We selectively discuss two successful industrial projects employing application generators in the construction of complex software, as representatives of the current state of practice.

- The SciNapse system [3] (formerly called Sinapse [18]) is a generator for mathematical modeling software. SciNapse uses both programmatic and pattern-based transformations and performs algorithmic refinement by using algorithm schemas, which are later specialized extensively. The specializations typically are numerical approximations for discrete representations of the continuous specifications of variables. SciNapse also includes transformations for data structure refinement and optimizations oriented towards scientific computing. The transformation process can be either automatic or interactive, with the user being able to override the system's choices at key points. The system is implemented in the Mathematica programming environment and uses Mathematica's algebraic manipulation capabilities. The system has multiple back-ends, generating code in Fortran 77, CM Fortran, or C.

  SciNapse was used originally to generate programs that solve partial differential equations for sonic wave modeling. These programs have multiple applications in exploring seismic wave propagation between oil wells, measuring the transit time of sonic waves in a moving fluid, exploring the 3D effects in complex geological formations, etc. More recently, the system was applied to financial modeling. SciNapse generates 200-4000 lines of code programs from compact (~50 lines) specifications. The generated programs exhibit performance often comparable to hand-coded versions and are commonly used with only small manual modifications.
- Mousetrap is a transformation system developed at Motorola, which has been applied to the derivation of efficient real-time code for the company's subscriber radio products [15]. Mousetrap operates on an abstract syntax tree intermediate form with fine-grained pattern-based transformations (tens to hundreds of thousands of transformations may be applied in the derivation of a complex system). The system performs algorithm selection based on algorithm schemas—e.g., translating a finite-state machine specification into code containing nested loops and conditionals. Multiple optimizations are applied in the generated code—for instance, loop invariant code motion, as well as machine-architectural optimizations like grouping bit-operations together and applying them at a machine-word level.

  The primary application of Mousetrap has been in generating *marshalling* code for subscriber radio protocols. The role of such marshalling code is to convert data from an in-memory representation (optimized for fast access) to the representation needed for wireless transmission (optimized for size). A set of Mousetrap transformations implement a gener-

ator for a domain-specific language that is used to describe the general structure of protocol packets. Because of optional information, many configurations of protocol packets can exist (all with the same general structure), and the transformation rules ensure that efficient code is created in every occasion. Many of the optimizing transformations employed in this process are domain-independent and part of the general Mousetrap infrastructure.

The result of generating marshalling code using Mousetrap has been "a tremendous success" [15]. The process was estimated to result in a reduction of the development cycle for marshalling code by a factor of four. Benefits in the maintainability and ease of code evolution were also observed.

5.2 Outlook

Generators are gaining momentum in the software engineering community. In the past few decades, software construction has not seen any radical improvements with respect to increased productivity and reliability. The proponents of the generator approach consider generators to offer the greatest promise among emerging software technologies for the future of software development. In particular, advocates of generators consider them to be the right tool every time a software product is designed to be reused, or every time a domain exhibits significant systematic variability. This view promotes generators as a substitute for most, if not all, of the existing software libraries for appropriate domains.

There are certainly serious challenges in trying to move generators to the forefront of software development. After all, generators are by nature domain-specific. Envisioning them as primary tools in the software construction process seems somewhat paradoxical. One possible scenario sees generators as libraries of coordinated transformations in a general transformation system. In this way, domain experts can be employed to build generators but any user of the transformation system can integrate them in a development process. According to this scenario, customers will be able to purchase and use domain-specific language implementations as these are needed for a particular project. For instance, a customer may base an application on a language module for the domain of windowing applications and another for the domain of networking. Both modules will be easily integrated as simple extensions to the functionality of the transformation system. The development environment (editor, debugger, etc.) will only change incrementally as a result of the integration.

Clearly, for this vision to take place, several issues need to be addressed. These vary from the technical to the social (organizational inertia and vested interests, for instance). Significant effort is being made in the (better defined) technical field. Indicatively, we mention the large-scale efforts on two promising transformation systems that are under development.

- The *Intentional Programming (IP)* system [2][41] is an ambitious, long-term project of Microsoft Research. IP's goal is to offer an advanced transformation environment with a high level of engineering maturity (e.g., modern amenities like debugging across layers of abstraction and editing support). The system's main *conceptual* contribution is a translation engine that schedules transformations based on dependencies that are discovered dynamically (that is, while transformations are being applied). The translation engine is fully backtracking: transformations that have been performed out-of-order can be rolled back and re-applied after all their dependencies are satisfied. Early versions of the engine have been documented [2] and the system has already been used to build generators [42].
- The *Design Maintenance System (DMS)* is a transformation environment for manipulating programs represented as flow graphs. DMS is based on the idea that the most valuable result of a software development effort is not the produced code, but the design information [8]. Thus, DMS stores design decisions in the form of choices of transformations and the reasons that led to those choices. Changes in the implementation are expressed as *maintenance deltas* and are (semi-)automatically integrated in the final software artifact. A maintainer of the software project can add maintenance deltas at any level of abstraction, and the system will evolve the new design and derive an updated implementation incrementally. One early product developed on the DMS transformational platform is the Clone Doctor (for C, C++, Java, and Cobol programs): a utility for reliably detecting code fragments with similar functionality and abstracting them into a single entity [9].

Both of the above systems demonstrate the ongoing work in the development of generator infrastructure. Armed with modern technical support, generators can play a prominent role in the future of software development.

6 References

[1] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[2] W. Aitken, B. Dickens, P. Kwiatkowski, O. de Moor, D. Richter, and C. Simonyi, Transformation in Intentional Programming, *5th Int. Conf. Softw. Reuse (ICSR 98)*: 114-123, 1998.

[3] R.L. Akers, E. Kant, C.J. Randall, S. Steinberg and R.L. Young, SciNapse: a problem-solving environment for partial differential equations, *IEEE Computational Science & Engineering*, 4(3): 32-42, July-Sep 1997.

[4] P.G. Bassett, *Framing Software Reuse: Lessons from the Real World*, Yourdon Press Computing Series, Prentice Hall, Upper Saddle River, NJ, 1996.

[5] D. Batory and J. Thomas, P2: a lightweight DBMS generator, *J. of Intelligent Information Systems*, 9: 107-123 (1997).

[6] D. Batory, B. Lofaso, and Y. Smaragdakis, JTS: Tools for Implementing Domain-Specific Languages, *5th Int. Conf. Softw. Reuse (ICSR 98)*: 143-153, 1998.

[7] D. Batory, G. Chen, E. Robertson, and T. Wang, Design wizards and visual programming environments for GenVoca generators, to appear *IEEE Trans. Softw. Eng*.

[8] I.D. Baxter, Design maintenance systems, *Communications of the ACM* 35(4): 73-89, April 1992.

[9] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, Clone detection using abstract syntax trees, *Int. Conf. Softw. Maintenance*, 1998.

[10] A. Berlin and D. Weise, Compiling scientific code using partial evaluation, *IEEE Computer*, 23(12): 25-37, Dec. 1990.

[11] T.J. Biggerstaff, Anticipatory optimization in domain-specific translation, *5th Int. Conf. Softw. Reuse (ICSR 98)*: 124-133, 1998.

[12] L. Blaine and A. Goldberg, DTRE—a semi-automatic transformation system, in B. Moller (ed.), *Constructing Programs from Specifications*: 165-204, North-Holland, 1991.

[13] L. Coglianese and R. Szymanski, DSSA-ADAGE: an environment for architecture-based avionics development, *Proc. Advisory Group for Aeronautical Res. and Develop. (AGARD)*, 1993.

[14] N. Dershowitz and J. Jouannaud, Rewriting systems, *Handbook of Theoretical Computer Science*: 243-320, Elsevier Publishers, Amsterdam, 1990.

[15] P. Dietz, T. Weigert, and F. Weil, Formal techniques for automatically generating marshalling code from high-level specifications, in *1998 Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, FL, October, 1998.

[16] J. Heidemann and G. Popek, File system development with stackable layers, *ACM Trans. on Computer Systems*, March 1993.

[17] N.C. Hutchinson and L.L. Peterson, The *x*-kernel: an architecture for implementing network protocols, *IEEE Trans. Softw. Eng.*, 17(1): 64-76, January 1991.

[18] E. Kant, *Synthesis of mathematical-modeling software*, IEEE Software, 10(3): 30-41, May 1993.

[19] U. Kastens, B. Hutt, and E. Zimmermann, *GAG: A Practical Compiler Generator*, Lecture Notes in Computer Science 141, Springer-Verlag, 1982.

[20] R. Kieburtz, L. McKinney, J. Bell, J. Hook, A. Kotov, J. Lewis, D. Oliva, T. Sheard, I. Smith and L. Walton, A software engineering experiment in software component generation, *Int. Conf. on Softw. Eng. (ICSE)*, 1996.

[21] D.E. Knuth, Semantics of context-free languages, *Mathematical Systems Theory*, 2:127-146, 1968.

[22] D.E. Knuth and P.B. Bendix, Simple word problems in universal algebras, *Computational Problems in Abstract Algebra*, J. Leech ed.:263-297, Pergamon, 1970.

[23] H.R. Lewis and C.H. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, 1981.

[24] C. Lin and L. Snyder, ZPL: an array sublanguage, *Languages and Compilers for Parallel Computing*, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds, 1993, 96-114.

[25] Y.A. Liu and T. Teitelbaum, Systematic derviation of incremental programs, *Science of Computer Programming*, 24(1): 1-39, February 1995.

[26] S.S. Muchnick, *Advanced Compiler Design and Implementation*, Morgan-Kaufman, 1997.

[27] J. Neighbors, *Software Construction Using Components*, Ph.D. Thesis, ICS-TR-160, University of California at Irvine, 1980.

[28] J. Neighbors, *Draco 1.2 Users Manual*, Department of Information and Computer Science, University of California at Irvine, June 1983.

[29] J. Neighbors, Draco: a method for engineering reusable software components. In T.J. Biggerstaff and A. Perlis (eds.), *Software Reusability*, Addison-Wesley/ACM Press, 1989.

[30] G. Novak, GLISP: a Lisp-based language with data abstraction, *A.I. Magazine*, 4(3): 37-47, Fall 1983.

[31] G. Novak, Generating programs from connections of physical models, *10th Conf. on A.I. for Applications (CAIA-94)*: 224-230, 1994.

[32] R. Paige and S. Koenig, Finite differencing of computable expressions, *ACM Trans. Programming Languages and Systems*, 4(3): 402-454, July 1982.

[33] R. Paige and J.T. Schwartz, Expression continuity and the formal differentiation of algorithms, *4th ACM Symp. on Principles of Programming Languages*: 58-63, January 1977.

[34] H.A. Partsch and R. Steinbrueggen, Program transformation systems, *ACM Computing Surveys*, 15(3), September 1983.

[35] H.A. Partsch, *Specification and Transformation of Programs: a Formal Approach to Software Development*, Springer-Verlag, Berlin Heidelberg, 1990.

[36] Reasoning Systems, *Dialect User's Guide*, Palo Alto, California, 1990.

[37] Reasoning Systems, *Refine 3.0 User's Guide*, Palo Alto, California, 1990.

[38] T.W. Reps and T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer-Verlag, New York, 1988.

[39] E. Schonberg, J.T Schwartz, M. Sharir, An automatic technique for selection of data representations in SETL programs, *ACM Trans. on Programming Languages and Systems*, 3(2): 126-143, April 1981.

[40] M. Sharir, Some observations concerning formal differentiation of set theoretic expressions, *ACM Trans. on Programming Languages and Systems*, 4(2): 196-226, April 1982.

[41] C. Simonyi, The death of computer languages, the birth of Intentional Programming, *NATO Science Committee Conference, 1995*.

[42] Y. Smaragdakis and D. Batory, DiSTiL: a transformation library for data structures, *USENIX Conf. on Domain-Specific Languages (DSL 97)*, 1997.

[43] D.R. Smith, Derived preconditions and their use in program synthesis, *6th Conf. Automated Deduction*, Lecture Notes in Computer Science 138, D.W. Loveland, Ed. : 172-193, Springer-Verlag, Berlin, 1982.

[44] D.R. Smith, KIDS: a semiautomatic program development system, *IEEE Trans. Softw. Eng.*, 16(9): 1024-1043, September 1990.

[45] D.R. Smith, KIDS: a knowledge-based software development system, in M.R. Lowry and R.D. McCartney (eds.), *Automating Software Design*, AAAI Press/MIT Press, Menlo Park, CA, 1991.

[46] D.R. Smith and M.R. Lowry, Algorithm theories and design tactics, *Science of Computer Programming*, 14(2-3): 305-321, 1990.

[47] D.R. Smith and E.A. Parra, Transformational approach to transportation scheduling, *Knowledge-Based Softw. Eng. Conf.*, 60-68, 1993.

[48] Y.V. Srinivas and R. Jullig, SPECWARE: formal support for composing software, *Conf. Mathematics of Program Construction*, 1995.

[49] G. Steele Jr., *Common Lisp: The Language*, Digital Press, 1990.

[50] A. Stepanov and M. Lee, The standard template library, in *ANSI/ISO Standards Committee C++ Standard*.

[51] D.S. Wile, *POPART: Producer of Parsers and Related Tools*, USC/ISI, November 1993.