Static Analysis for Program Generation Templates[1]

Valdis Berzins
Naval Postgraduate School
Monterey CA 93943 USA

Abstract

This paper presents an approach to achieving reliable cost-effective software via automatic program generation patterns. The main idea is to certify the patterns once, to establish a reliability property for all of the programs that could possibly be generated from the patterns. We focus here on properties that can be checked via computable static analysis. Examples of methods to assure syntactic correctness and exception closure of the generated code are presented. Exception closure means that a software module cannot raise any exceptions other than those declared in its interface.

1. Introduction

Our goal is to provide cost effective means for creating reliable software. We are addressing the issue by improving the technology for automatic software generation, with particular attention to reliability issues.

We take a domain specific view of this process: a domain is a family of related problems addressing a common set of issues. A domain analysis identifies the problem and issues, formulates a model of these, and determines a corresponding set of solution methods. Users of the proposed computer-aided software generation system describe their particular problem using a domain specific problem modeling language that provides concrete representations of problems in the domain. The system then automatically determines which solution methods are applicable, customizes them to the specific problem instance described using the modeling language, and then automatically generates a program that will solve the specified problem.

We seek to provide tool support for the above process that can be applied to many different problem domains, and that can generate code in any programming language. Therefore we seek uniform and effective methods for generating software generators of the type described above, given definitions of the problem modeling language, the target programming language, and the roles for synthesizing solution programs. A simple architecture for this process is shown in Figure 1.

The specific goals of this paper are: (1) to provide a simple example of a language for expressing software patterns that are specific enough to be used as synthesis rules and (2) to provide examples of static rules in this language. We address the problems of certifying that all programs which can be generated from a given set of rules: (1) are syntactically correct and (2) will not raise any exceptions other than those explicitly specified in an interface description.

This is a step towards a coordinated system of static and dynamic checks, to be performed on program synthesis rules. Our hypothesis is that the most cost effective way to improve software quality is to systematically improve and certify the rules used to generate a domain-specific software generator. This approach directly addresses the issue of correctly implementing given software requirements. It also indirectly addresses the issue of getting the right requirements, because it should eventually enable rapid prototyping of product quality systems by problem domain experts, who need not be software experts. If the requirements are found to be inappropriate, the domain experts will simply update the problem models and regenerate a new version of the solution software.

We will refer to the software generation patterns as templates. Our rationale for the claim of cost effectiveness is that the benefits of quality improvements to the templates can be extended to all past and future applications of the generators - by regenerating the generator using the improved templates and then regenerating the past applications. The regeneration process can be completely automated, thereby reducing labor costs, eliminating a source of random human errors, and speeding up the process of repairing a known fault throughout a large family of software systems.

[1] This research was supported in part by the U. S. Army Research Office under contract/grant number 35037-MA and 40473-MA, and in part by DARPA under contract #99-F759.

The relation to the theme of this workshop is that fast moving scenarios can be addressed by automatically generating new variants of the software that reflect changing issues in the problem domain. Our approach should reduce the explicit quality assurance efforts needed each time the software is changed. By amortizing the quality assurance effort applied to the template over many applications of the same templates, we can reduce quality assurance costs. The benefits increase with the number of systems generated from the same templates.

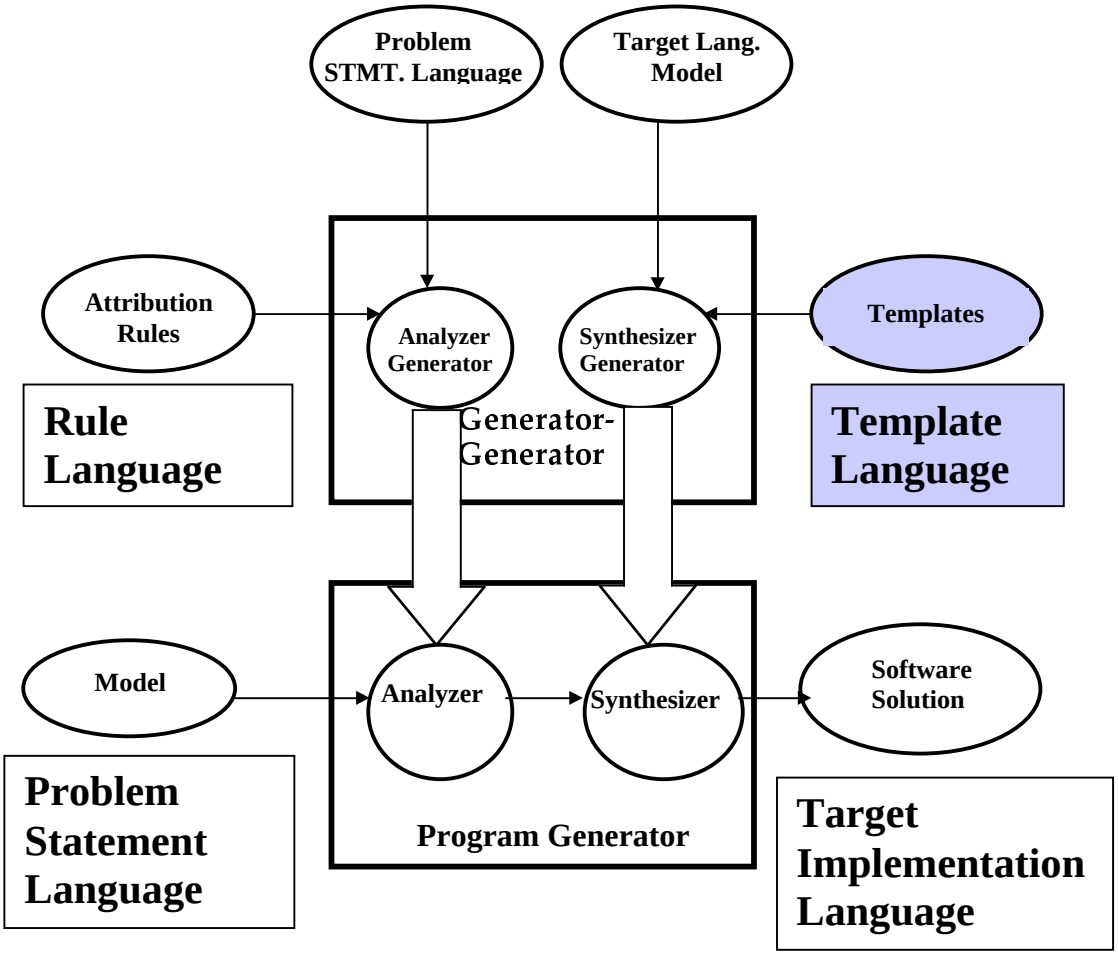

Figure 1. Model-Based Software Generator Architecture

This paper focuses on static checks that can be completely automated. Our research is also addressing testing and debugging of program synthesis rules and proofs of rule properties that require human assistance with deeper reasoning. These efforts are outside the scope of the current paper, which is organized as follows:

- Section 2 formalizes software generation patterns and defines a uniform construction to obtain a template language for any target programming language.
- Section 3 describes methods for statically certifying syntactic correctness generated code, and gives an example.
- Section 4 does the same for analysis of exceptions.
- Section 5 contains comparisons to previous work
- Section 6 presents conclusions.

2. Template Languages

The purpose of a template language is to define software synthesis patterns for a given target language. We create such languages based on a functional object model of code generation templates. We take a functional (i.e. side-effect-free) approach because this simplifies the algebraic basis of the approach and supports effective static analysis methods such as those presented in Section 3 and 4.

We view template languages as extensions of the corresponding target programming languages. Because many different programming languages are created, we will need many different template languages. However, all of these can be defined at once by providing uniform construction such as that shown in Figure 2.

This is a very simple construction, but it is very expressive. In addition to providing substitution of actual values for generic parameters, as in the generic units of Ada and the templates of C++, our construction includes conditionals that are evaluated at code generation time, and the ability to invoke other templates. Recursion is included.

```
Template_language = {template, formal_def, template_expression}

DEF_TEMPLATE(id[template], type, seq[formal_def], template_expression):
        template      -- where type ° ∈ target_language

DEF_FORMAL(template_parameter, type): formal_def
        -- declares the type of a formal parameter

template_parameter < {id[any], template_expression}
IF(template_expression, template_expression, template_expression):
        template_expression
APPLY(id[template], seq[template_expression]): template_expression

template_expression < target_language
```

Figure 2. Template Abstract Syntax

The construction depends heavily on the use of inheritance in object-oriented modeling of programming languages. The situation is illustrated in Figure 3.

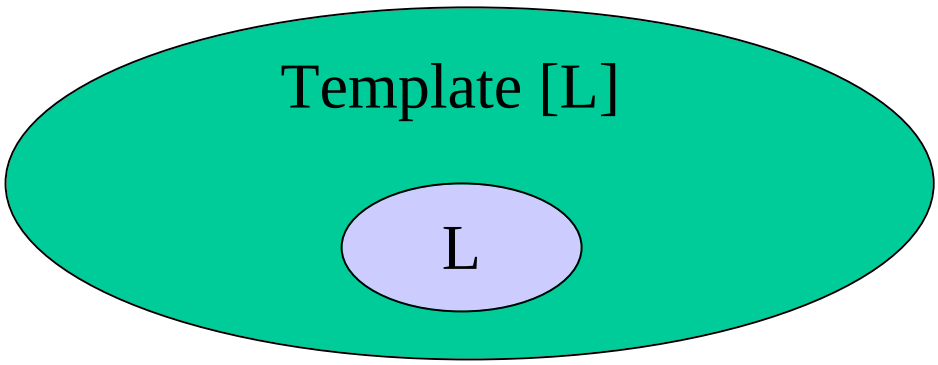

Figure 3. Generic Template Language

In object-oriented modeling, class-wide types[2] are viewed as open and extensible. Specifically, each time we add a subclass with a new constructor, we add more instances to the class-wide type, thus extending its value set.

We model the abstract syntax of a language using a type for each kind of semantic entity. In a properly constructed abstract syntax, there should be one such type for each non-terminal symbol. Each constructor of these types corresponds to a production of the grammar. Subclass relationships, denoted by "≤", specify that every instance of the subclass is also an instance of the parent class. Multiple inheritance is allowed. For example, in line 6 of Figure 2 says that every template parameter is a kind of identifier, and also is a kind of template expression. This kind of subclass relationship is used to incorporate reusable types in a library of programming language building blocks, such as identifiers, and to specialize reusable concepts to the application, such as template expression. If T is a type and S is a set of types, T<S means T is a subclass of each element of S. This represents multiple inheritance.

[2] This is Ada 95 terminology. The instances of a class wide type include its direct instances and those of all its subclasses, transitively.

Subclassing is also used to interface between a target programming language and its extensions. In Figure 2, "target-language" denotes the set of types comprising the abstract syntax of the target language. Figure 4 shows a very simple example of a target language that illustrates how this works.

```
target_language = (stmt, exp)

assign(var, exp): stmt
if(exp, stmt, stmt): stmt

integer < exp     -- integer literals
var < {id[any], exp} -- program variables
apply(id[function], seq[exp]): exp -- operations

subtype rule:  x < y ==> id[x] < id[y] where x, y ∈  type
```

Figure 4. Example: Micro Target Language

The example in Figure 5 defines a code generation pattern that embodies Newton's method for polynomial evaluation, which is optimal in terms of number of evaluation steps needed. This is a very simple example of a code generation pattern that is nevertheless realistic, because it embodies a solution method. The example also illustrates the use of all the constructs in the template language. We use infix syntax for the exp constructors * and + to improve legibility (e.g. x*y is short for the term apply(*, x, y)).

An additional benefit of considering the abstract syntax to be an algebra rather than a tree is that we can used well-studied transformation rules. In particular we can associate equational axioms with the programming language types that define normal forms. Figure 5 illustrates the use of such axioms as rewrite rules that simplify the code produced by the generator in a follow-on normalization process. This is one way to incorporate optimizations into the program generation process, which is useful for unconditional transformations.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp
    -- c contains coefficients of a polynomial, lowest degree first
    IF not (is_empty (c) )   -- use operations of boolean and seq
    THEN v * (evaluate_polynomial (v, rest(c))) + first (c)
    ELSE 0
END TEMPLATE

Template application evaluate-polynomial(x, [1, 2, 3]) generates
      x * (x * (x * 0 + 3) + 2) + 1

Normalization with integer rules i * 0 = 0, i + 0 = i reduces to
      x * (x * 3 + 2) + 1
```

Figure 5. Example: Generation Pattern

Code generation using the template language is a very much like evaluation in a functional programming language with call-by-value semantics. Analysis of templates can take advantage of equational reasoning, substitution, and structural induction. The limitation to primitive recursion facilitates the latter. The recursion in the example is structural because **rest** is a partial inverse for the sequence constructor **add** (i.e. rest(add(x, s)) = s).

3. Syntactic Correctness of Generated Code

We treat the abstract syntax structures of the target language as the values of the abstract data types representing the programming language. We require these types to provide a pretty printing operation that outputs such objects as text strings according to the concrete syntax of the target language, with a readable format. Establishing correctness of these pretty printing operations is straightforward, and in fact their implementations can be generated from an appropriately annotated grammar for the concrete syntax.

Given trusted pretty printing operations for the object model of the target language, syntactic correctness of the output reduces to the type-correctness of the ground terms generated by the evaluation

of the templates. This can be checked using a simple type system for the template language and conventional type checking methods. Note that we are referring to the types associated with the signatures of the constructors in the object model of the target programming language, rather than the types within the target programming language, which may not even be a typed language. The process is illustrated Figure 6. The computed type annotations are shown in *italics*. The type annotations associated with the implicit induction step, where the type signature of the template itself is used, is highlighted in ***bold italics***. The indentations of the type annotations reflect the structure of the derivation.

TEMPLATE evaluate_polynomial (**v**: var, **c**: seq[integer]): exp
  IF not (is_empty (c *: seq[integer]* **)** *: boolean* **)** *: boolean*
  THEN + (* (v *: var,*
  evaluate_polynomial
  (v ***: var*** **,**
  rest(c: seq[integer]) ***: seq[integer]*** ***: exp***
) *: exp*
  first (c: seq[integer]) *: integer*
) *: exp*
  --term form of **v* evaluate_polynomial (v, rest(c)) + first (c)**
  ELSE 0 *: integer*
END TEMPLATE

Types conform because `integer < exp` and `var < exp`

Relevant signatures: +(exp, exp) :exp, *(exp, exp) :exp,
first(seq[T]): T, rest(seq[T]): seq[T],
is_empty(seq[T]): boolean, not(boolean): boolean

Figure 6. Example: Syntactic Correctness of Generated Code

Note that induction has been carried out implicitly, as a routine step of the type checking calculation. This is sufficient to establish partial type correctness of the templates, which implies syntactic correctness of all code that could be generated by the template, it does not automatically guarantee total correctness, because we still have the possibility that evaluation of the template might fail to terminate.

Total correctness is established by the type check if we check that all recursions are primitive. The example satisfies this condition because rest is a partial inverse of the compound sequence constructor; rest(add(x,s)) = s. This means that the induction is in fact structural, and hence that evaluate_polynomial is total. Thus the template will produce syntactically correct code for all input values that conform to the type signature of evaluate_polynomial.

We note that given declarations of the target language constructors that define the abstract syntax and the corresponding partial inverse operations, it is straightforward to automatically check that all recursive calls are primitive with respect to any given parameter position. This implies that structural induction can be applied uniformly and completely automatically in this context. Furthermore, our experience suggests that structural recursions are sufficient to define the code generation templates needed in practice, and that template designers can live within the restriction to structural recursions without undue hardships.

4. Exception Closures for Generated Code

One common source of software failure is unhandled exceptions. This section explains a method for certifying that all programs generated from a given template cannot generate any unhandled exceptions when placed in a context that handles a specified set of exceptions.

Our approach is to refine the type system to record the set of exceptions that might be raised by the evaluation of any expression of the target language. A similar structure can be used to analyze the set of exceptions that might be raised by execution of a statement of the target language.

The refinement replaces the single target language type exp with a parameterized family of types exp[set[exception]]. The intended interpretation of this type structure is that evaluation of an expression of type exp[S] might raise an exception e only if e∈ S. Since we do not require all exceptions in S to be producible, this family of types has a rich subclass structure defined by the following relation:

$$S1 \subseteq S2 \Rightarrow \text{exp}[S1] \leq \text{exp}[S2]$$

The type signatures of an operation are specified explicitly for argument expression type that cannot raise any exceptions, and are extended to all other types by the following rule, which describes the essential pattern for propagating exceptions:

$$F(\text{exp}[\emptyset]) : \text{exp}[S1] \Rightarrow f(\text{exp}[S2]): \text{exp}[S1 \cup S2]$$

The rule for operations with multiple arguments is similar. Similar rules apply to language constructs representing exception handlers. Exception handlers follow rules of the form

$$(\text{TRY exp}[S1] \text{ CATCH e USE exp}[S2]): \text{exp}[(S1-\{e\}) \cup S2].$$

Figure 7 shows the exception analysis for our running example. The parts added to the version in Figure 6 are underlined.

```
TEMPLATE evaluate_polynomial (v: var, c: seq[integer]): exp [{ovfl}]
 IF not (is_empty (c: seq[integer]): boolean): boolean
    THEN +(*(v: var,
             evaluate_polynomial(v: var,
                rest(c: seq[integer]): seq[integer] ): exp [{ovfl}]
             first (c: seq[integer]): integer): exp [{ovfl}]
     -- term form of  v * evaluate_polynomial (v, rest(c)) + first (c)
    ELSE 0: integer
END TEMPLATE
```

Types conform because integer < exp[∅] ≤ exp[{ovfl}] and
var < exp[∅] ≤ exp[{ovfl}]

Relevant signatures: +(exp, exp): exp [{ovfl}], *(exp, exp): exp [{ovfl}],
first(seq[T]): T, rest(seq[T]): seq[T], is_empty(seq[T]): boolean, , not(boolean): boolean

Figure 7. Exception Closure of Generated Code

Note that we require the author of the template to specify in the type declaration of a template the set of exceptions the generated expression is allowed to raise. This acts as an induction hypothesis in our exception analysis, which is used when analyzing the recursive call of evaluate-polynomial. It also provides useful information for the user of the generated code.

The analysis shown in the figure establishes a partial exception closure: it guarantees that all expressions generated by the template can at most raise only the exception **ovfl** representing integer overflow.

To establish a total exception closure, we have to address clean termination of the template expansion at program generation time. The primitive recursion check explained in the previous section guarantees there will be no infinite recursions, so that termination is guaranteed. However, for clean termination, we must also check that evaluation of the template will not raise any exceptions at program generation time.

Note that the analysis in Figure 7 addresses run-time exceptions. When viewed as constructors of the abstract syntax, + and * are total operations. Overflow exceptions can occur only when those expressions are evaluated, not when they are constructed.

The sequence operators **first** and **rest** are different: they are partial query methods of the abstract syntax, not total constructors. If applied to an empty sequence, they raise a sequence underflow exception. However, this can occur only at program generation time, not at run time.

To certify clean termination of template at program generation time requires a type refinement to record sets of possible exceptions and an additional kind of type refinement to record domains of partial methods such as **first** and **rest**. We can introduce a subtype nseq[T, S] < seq[T, S] consisting of the nonempty sequences, and refine the signatures of the partial sequence operations first and rest as follows.

first(nseq[T, ∅]): T[∅], rest(nseq[T, ∅]): seq[T, ∅]
first(seq[T, ∅]): T[seq_underflow], rest(seq[T, ∅]): seq[T, {seq_underflow}]

Type analysis requires a bit of inference in this case, because we have to use the guard of the template language conditional IF together with the rule

s : seq[T, S] and not is-empty (s) ⇒ s: nseq[T, S]

This inference is easy because the guard matches the subtype restriction predicate for nseq[T].

This match did not occur by accident - the purpose of the guard is precisely to ensure that the operations first and rest are used only within their domain of definition. In the interests of being able to produce certifiably robust code, we claim that it would not be unduly burdensome to require that template designers associate domain predicates with all partial operations, and use those domain predicates explicitly in guards whenever they are needed to ensure the partial operators are used within their proper domains of definition. For example, **first** could be associated with a domain predicate

first-ok (seq[T]) : boolean where
first-ok (s) = not (is-empty (s)).

This would enable a fast and shallow analysis of guard conditions to certify absence of exceptions in cases like this. Some such restriction is necessary for practical engineering support because the problem of checking whether an unconstrained guard condition implies the domain predicates of arbitrary guarded partial operations is undecidable.

An alternative is an exception analysis that includes exceptions in the closure even in cases where the guard condition ensures they will never arise. We suggest that it is more practical to handle a common subset of efficiently recognizable forms, and to ask designers to work within the constraints of those recognizable forms. We believe this would be less burdensome than the alternative of manually analyzing the cases where a type check insensitive to guard conditions would nominate exceptions that cannot in fact occur, and that it would lead to a more robust software by making it practical to do complete analysis of exception closures. For example, we could require the example of Figure 7 to be written in a stylized form that looks like the following:

IF first-ok (c) and rest-ok (c)
THEN ... **first** (c) ... **rest** (c) ...

A similar type check would have to be applied to the implementations of **first** and **rest** to ensure that they would in fact terminate cleanly whenever the domain predicates are true.

5. Comparisons to Previous Work

One of our contributions has been to formalize and abstract the idea of a program generation pattern, to make it independent of the details of the target programming language and the process of instantiating the patterns. The purpose of this was to create context in which systematic analysis of program generation patterns becomes possible and in some cases becomes decidable.

Program generation patterns have been evolving for a long time. Macros are an early form of the idea. However, macros are notoriously difficult to analyze, partially because they traditionally operate on uninterpreted text. This makes the connection between macro definitions and the behavior they ultimately denote complicated and potentially very indirect. The macros in LISP are an improvement because they are based on abstract syntax trees rather than characters. However, in this context a second source of complexity becomes apparent: a macro can expand to produce another macro, and the number

of expansion steps before the generated source code actually appears is potentially unbounded. This makes the system very difficult to analyze. At the other extreme are the generic units of Ada. These are strongly typed, clearly connected to the abstract syntax of the language, and the results of instantiating them are easy to analyze. However, they do not allow conditional decisions at instantiation time, and are restricted in the sense that the abstract syntax trees of all possible instantiations have exactly the same shape, up to substitution for the formal parameters of the pattern. A language-independent version of the idea can be found in [5], although this appears to be largely text-based.

Another aspect of our approach is to model languages as algebras rather than as abstract syntax trees. A hint of this idea appears in [4], although it is not exploited there for enabling analysis to any significant degree. The work of the CIP group [1] develops this idea further and takes advantage of the reasoning structures that come with the algebraic modeling approach, such as term rewriting and generation induction principles. This suggests extension to a full object-oriented view, which includes inheritance. The Refine system is the earliest context we know of where grammars are treated as object models with potential inheritance structures, although the documentation does not give any hint about the significance of this capability. In this paper we demonstrate the usefulness of algebraic models of syntax with inheritance, for defining language extension transformations that can be applied to all possible target languages.

Another theme is lightweight inference [2]. We have demonstrated that some useful types of static analysis for program generation patterns can be performed via computable and indeed reasonably efficient methods. The processes described here can be implemented using technologies typically used in compilers, such as object attribution rules, they terminate for all possible inputs, and do so in polynomial time. We believe this approach will scale up to large applications, and are currently working out the details to support a tight analysis of the efficiency of the process.

This paper has explored static analysis of meta-programs to check syntactic correctness and exception closure of the generated code. Another kind of static analysis in this family, type checking of meta-programs to ensure the type correctness of the generated code, is considered by another paper in this proceedings [3].

6. Conclusions

We believe that formal models of program generation templates can support a variety of quality improvement processes that can help achieve cost-effective software reliability. This paper has presented a simple example of such a formal model and two such quality improvement processes, certification of syntactic correctness and freedom from unexpected exceptions for all programs that can be generated from a given program generation pattern. We expect the greatest advantages of this approach to be realized when it is applied to realize flexible and reliable systems in a product line approach. This approach should be augmented with systematic methods for domain analysis that culminates in the development of a domain-specific library of solutions embodied in a domain-specific software architecture that is populated with components produced by model-based software generators. When the technology matures, it should become possible for problem domain experts to specify their problem instances in terms of familiar problem domain models, and to have reliable software solutions to their problems automatically generated, without direct involvement of computer experts.

The economic advantage of this approach comes from the ability to automatically reap the benefits of each quality improvement for all past and future instantiations of the template (if past applications are regenerated). We believe that it will be profitable to explore methods for lifting many known program analysis techniques from the level of individual programs to the level of program generation patterns. This should be explored for a variety of issues that range from certifying absence of references to uninitialized variables, absence of deadlock, and many others, perhaps ultimately to template-based proof of post conditions and program termination for generated programs.

To make this vision practical, many engineering issues must be addressed, including presentation issues, methods for lightweight inference [2] and support for transforming and enhancing complex sets of analysis rules. Other issues include systematic methods for dynamic analysis, testing, and debugging of program generation rules. It is not reasonable to expect progress to occur in an instantaneous quantum leap to perfection. A realistic process is a gradual one, where simple sets of program generation rules are deployed, and gradually tuned, improved, certified, and extended. A key issue is enabling rule enhancement and exception closure extension without invalidating all previous effort on analysis and certification of the previous versions.

The difference between the program generation approach proposed here and current compiler generation tools is the associated static analysis capabilities for the program generation rules. It is possible that in the future, ultra-reliable compilers will be built using techniques derived from those introduced in this paper.

REFERENCES

1. F. Bauer, H. Ehler, A. Horsch, B. Moller, H. Partsch, O. Paukner and P. Pepper, *The Munich Project CIP*. Vol. 2: The Program Transformation System CIP-S, Springer, Berlin, 1987.

2. V. Berzins, ÒLight Weight Inference for Automation EfficiencyÓ Proceedings of the *1998 ARO/ONR/NSF/DARPA Monterey Workshop on Engineering Automation for Computer Based Systems*, Monterey California, 1999.

3. N. Bjorner, ÒType Checking Meta ProgramsÓ Proceedings of the Workshop on *Modeling Software System Structures in a Fastly Moving Scenario*, Santa Margherita, Italy, 2000.

4. T. Reps, *Generating Language-Based Environments*, Doctoral Dissertation, August 1982.

5. D. Volpano, R. Kieburtz, ÒSoftware TemplatesÓ CS/E 85-011, Department of Computer Science and Engineering, Oregon Graduate Center, 1985.