Executable Specifications for Language Implementation

Uwe Kastens

Fachbereich Mathematik/Informatik
University of Paderborn, D-4790 Paderborn, F.R. Germany

Abstract. Generating programs from specifications is an ambitious task that is solved at most for restricted application domains. General solutions which are practically satisfying as well are hard to achieve. Language implementation is a field, where tools and toolsets are available which process executable specifications and derive language implementing programs (compilers or interpreters) from them.
In this paper we will study specification principles that contribute to the success of program generation in this application domain. Examples are taken from the specification techniques of the Eli-System. The task of language implementation is decomposed into subtasks which have well understood and sufficiently general algorithmic solutions. Hence the instances of subtasks for a particular language can be specified. Certain language concepts like scope rules can be understood as a combination and variation of some basic rules. This situation allows specifications on a rather high level and reuse of precoined solutions. Domain specific expert know-how embedded in a toolset can further raise the specification level. The presentation of such specification principles in language implementation may raise discussion whether and how they can be applied to other areas as well.

1 Introduction

In the reference manual of Z [8] Spivey characterizes formal specifications as follows:

> "Formal specifications use mathematical notation to describe in a precise way the properties which an implementation must have, without unduly constraining the way in which these properties are achieved. They describe *what* the system must do without saying *how* it is to be done."

The abstraction of the *what* from the *how* shall achieve specifications that have a small cognitive distance to the system requirements and a large distance to an implementation. Such specifications are *declarative* rather than *operational*.

Specifications have an important role in the software life-cycle: They are a reference point for both requirements analysis and implementation, and are a valuable means of promoting a common understanding among all those concerned with the system.[8] Specifications serve for proving an implementation against the requirements with respect to certain properties, e. g. invariants on

the system states, I/O relation of a function, or mutual exclusion of critical operations in a parallel system.

The role of specifications in software development is further increased if an implementation is derived by refinement of the specification. Each refinement step introduces a design decision moving towards an implementation while keeping the specified properties intact. If we could get to an implementation without augmenting the specification of system properties by design decisions explicitly we had an *executable specification*. It could serve either for prototyping or for generating software products, depending on the software quality of the implementation.

Executable specifications especially for rapid prototyping is the goal of specification languages classified as Very High Level Language (VHLL). Krueger [7] discusses VHLL like SETL, PAISLey, and MODEL under the aspect of software reuse. The reuse effect is achieved by the specification language compiler or interpreter. It makes the implementation decisions without involving the author of the specification.

General purpose specification languages, as those mentioned so far, are based on elementary mathematical concepts: sets, functions, and sequences for modelling data, and predicate logic for modelling properties of operations. Systems that interpret or compile such specifications have to use generally applicable implementation strategies. So on the one hand all aspects of a system have to be specified and refined down to those elementary concepts. On the other hand the efficiency of the automatically achieved implementation is at best acceptable for prototyping.

This situation can be dramatically improved if the problem domain is restricted to a certain application area: A system of that domain can be described in terms of a dedicated specification language. An *application generator* translates such a specification into an implementation [1]. Krueger [7] characterizes domains appropriate for application generators, if "many similar software systems are written, one software system is modified or rewritten many times during its lifetime, or many prototypes of a system are necessary to converge to a usable product". Report generators for data bases are a typical area for application generators [3].

Narrowing the problem domain yields important advantages for specification design and execution:

- A specification may refer to concepts that are well-understood in the domain and hence need not be further refined.
- A domain specific model for problem decomposition can induce a modular structure of the implementation without being specified explicitly for each system.
- Domain specific implementation techniques can be applied automatically.

Hence systems are described on a high level and the specifications are executable.

In this paper we discuss strategies for executable specifications in the domain of language implementation. Translator generation can also be considered as an

instance of the application generator principle, although this research is much older than the application generator idea. More than forty years of research and practice in compiler construction have resulted in common understanding of task decomposition and of subproblems, powerful formal methods for problem description, and in systematic implementation techniques. Tools are available that generate implementations from specifications, hence achieve their executability. The domain is very broad, ranging from compilers and source-to-source translators for programming languages to the implementation of dedicated specification languages, as used for application generators.

In the following sections we emphasize the discussion of specification strategies applied in this application domain to achieve executability. We use Eli [2] [4] as an example for a system which integrates many generating tools, precoined solutions, and domain specific knowledge. A major design goal of Eli is to achieve executable specifications that have a small cognitive distance to requirements of its problem area. We have learned many aspects of the specification strategies discussed here from the experience in developing and using the Eli system.

2 Domain Specific Decomposition

Decomposition of problems into subproblems is a natural method for analysis and design. Different aspects of a problem are separated and described on a suitable level of abstraction. Solutions can be found for smaller units using different techniques suitable for the particular subtask. Modular structure of the implementation and its interfaces can be derived from the decomposition structure.

If the problem space is restricted to a certain application domain specification and solution can be supported by a domain specific decomposition model that can be applied for any particular problem instance of that domain. Many years of experience in the language implementation domain led to a generally accepted model for decomposition of compilation tasks, as shown in Figure 1 taken from [2]. That model is not restricted to programming language compilers: In case of arbitrary language translation or interpretation the transformation phase usually yields the final result, the encoding phase is left out.

The existence of a suitable domain specific decomposition model simplifies the development of particular problem specifications: The description of the model leads to a structured way of reasoning about the problem - even if users are not experienced in language design and translation: It becomes obvious that, for example, the form of tokens of the input language has to be specified, or rules for name analysis must be chosen if the language has named objects. The model suggests that these properties of the problem refer to different subtasks and that they are related by the representation of name tokens.

A domain specific decomposition also allows one to specify different subtasks via dedicated formal models using suitable specification languages: E. g. the form of tokens is described by regular expressions like

```
Ident: $[a-zA-Z][a-zA-Z0-9]*
```

Structuring	Lexical analysis	Scanning Conversion
	Syntactic analysis	Parsing Tree construction
Translation	Semantic analysis	Name analysis Type analysis
	Transformation	Data mapping Action mapping
Encoding	Code generation	Execution-order determination Register allocation Instruction selection
	Assembly	Instruction encoding Internal address resolution External address resolution

Fig. 1. Compilation Subproblems

or the structure of the translated target text by a named pattern like

```
Block: "{" $1 $2 "}"
```

where the declarations are to be substituted at the first and the statements at the second pattern variable.

The decomposition of the problem applies also to a modular decomposition of the solution. Hence the model supports automatic derivation of subtask implementation. Well-understood formal methods can be used separately for solutions of different subtasks, e. g.

- finite automata for scanning,
- LALR(1) parsers with tree construction for structural analysis,
- attribute grammars for dependent computations in trees.

Separation of subtask specifications allows dedicated generating tools to be applied, each solving nothing more than its specific task. If subtasks like structuring and semantic analysis were not separated those implementation methods would not be automatically applicable.

Without a fixed decomposition model we would have to use a general purpose specification language, and would have to design a decomposition for each particular problem instance. It would then be almost impossible to deduce application of domain specific implementation methods automatically. On the other hand the decomposition model requires that the specification is refined down to the details of each subtask. That might not be desirable in each case. We shall consider this aspect in Section 4.

3 Integration of Tools

In the previous section we argued for having a domain specific decomposition model. Of course the specifications of subtasks are related and their implementations have interfaces. If tools are applied separately for each subtask, the relations

between components of the specifications are not taken into account. The user has to take care that the solutions fit together. On the other hand an integrated system that processes the complete set of specifications can control those relations, and direct the relevant information to particular tools. Furthermore the knowledge of domain experts can be embedded in the system such that most of the interfacing problems are solved without being specified by the user.

We demonstrate these aspects by tracing the relationship of subtasks between identifier tokens and scope rules for named objects. The decomposition model tells us that we have to deal with identifiers in the subtasks scanning, parsing and name analysis. Figure 2 shows these parts extracted from a complete specification. Part (a) describes the form of identifiers for scanning. `mkidn` refers to a function that yields a bijective mapping from strings to integral encodings. Part (b) shows a fragment of the context-free grammar specifying the input structure of programs for the parsing task. The attribute grammar fragment (c) specifies two distinguished contexts of the tree grammar where identifier tokens occur as leaf nodes. Their instrinsic attributes `Sym` are propagated one level up in the tree by the associated computations. Part (d) associates certain scope rule patterns for defining and applied identifiers to the particular phrases of this tree grammar (Section 5).

The specification fragments of Figure 2 are related to one another: The token class `Ident` is a terminal symbol of the concrete grammar, and it corresponds to a class of leaf nodes in the tree grammar. The encoding information derived from the token is used as an intrinsic attribute for computations in the tree, in this case by the name analysis computations of Figure 2 (d). An integrated system can check the completeness and consistency of the specifications with respect to such relations.

Furthermore redundancy of specifications can be reduced by deriving the specifications for single subtasks from the complete set of specifications. For example keywords, operators, and delimiters need not be specified separately for the scanning task; their form is derived from the grammar productions. An integrated system can even compute the mapping between productions of the concrete grammar and the tree grammar. That mapping may be more complex than the one-to-one mapping of the productions in Figure 2, e. g. for expressions with several precedence levels. In many cases productions of the one grammar can be derived from the other. In our example part (b) could be completely omitted, taking that part of the parser specification from (c).

It is well-known that tools can generate translator modules from specifications derived from those like that in Figure 2: implementations of a finite automaton for scanning, a LALR(1) stack automaton for parsing, and a tree walking algorithm for attribute computation in trees. These modules have to fit together in order to make up an executable translator. This goal can be achieved without describing any technical aspects of interfaces or implementation in the specification. An integrated system like Eli embeds domain specific knowledge on how to interface the modules and how to configure the executable product.

In case of our example several such engineering decisions are either fixed

```
Ident: $[a-zA-Z][a-zA-Z0-9]* [mkdin]
```

a) Identifier token specification

```
Decl   ::= Type VarDef ';'.
Assign ::= VarUse ':=' Expr.
VarDef ::= Ident.
VarUse ::= Ident.
```

b) Identifier terminals in the concrete grammar

```
RULE rDecl: Decl ::= Type VarDef ';' END;
RULE rAssign: Stmt ::= VarZse ':=' Expr ';' END;
ATTR Sym: int;
RULE rVarDef: VarDef ::= Ident COMPUTE
      VarDef.Sym = Ident.Sym;
END;
RULE rVarUse: VarUse ::= Ident COMPUTE
      VarUse.Sym = Ident.Sym;
END;
```

c) Identifiers in tree grammar contexts

```
SYMBOL VarDef INHERITS IdDefNest, IdDefUnique END;
SYMBOL VarUse INHERITS IdUseNest, NokeyMsg END;
```

d) Scope rules specification related to identifier occurrences

Fig. 2. Related specifications for named objects

for all language processors generated or they are derived from the particular set of specifications: A fixed interface is used between the scanner and modules that store and encode token representations, like the function **mkidn** of a symbol table module. A module for tree construction is generated from the tree grammar specification. The mapping between the tree grammar and the context-free input grammar determines when the tree construction functions are to be called by the parser. Appropriate actions are added to the parser generator input.

Hence on the base of a domain specific decomposition model implementation decisions on interfaces can be made by an integrated system without the need for explicit implementation level specifications.

4 Declarative Specifications

The solution of a particular problem P can be described by one of three general methods [9]:

1. specifying properties of the problem P,
2. identification of P with the description of a problem Q,
3. describing a solution of P.

Methods (1) and (2) lead to a declarative specification style. The description of a solution (3) is usually operational on the level of implementation. Hence it should be considered as an escape from declarative specifications for aspects not covered by the underlying formal model. E. g. in Figure 3.1 (a) regular expressions does not cover storing and encoding of identifiers by a symbol table. Association of a function call **mkidn** is an escape to an operational level. Most declarative specification techniques need such operational hooks. They either attach solutions to the specification which are supplied on implementation level, or they connect specifications using different models. The careful distinction between declarative and operational aspects supports clarity of the specifications and allows tools to strictly apply formal methods for the declarative part.

Figure 2 shows the use of specification languages for the description of the form of tokens, by regular expressions, syntactic structure by context-free grammars, computations in trees by attribute grammars. Tools generate executable program modules from such specifications. In general none of these subtasks are completely described in declarative style. For that purpose operational solutions of smaller subtasks can be hooked to the description of problem properties without destroying their declarative character. We give three examples for this aspect:

The name of a symbol table function **mkidn** is attached to the description of the identifier tokens. The scanner generator inserts a call of that function into the scanner code. The implementation of that function is either taken from a system library or it is supplied by an operational solution.

Similarly actions for tree construction are attached to context-free productions and processed by the parser generator. In this case those operational hooks are not visible on the level of the original specification, and their implementation need not be specified.

The tasks of the semantic analysis and the transformation phase are specified in the calculus of attribute grammars. Its declarative aspect associates computations (function calls) to tree contexts and states dependencies between them. The attribute grammar tool decides how to walk the tree, when to call the functions, and where to store attribute values. On the other hand the implementation of the functions and the representation of the attribute values lies outside of the declarative aspect of that calculus. They are either contributed by tools for other subtasks, e. g. a pattern driven generator for output functions, or operational solutions are taken from a library or given by the user.

The method (2) of describing a problem instance by identification with another problem that has a known solution is typical for domain specific applications. For example one could specify the form for identifier tokens by referring to those of the language C:

```
Ident: C_IDENTIFIER
```

Here the token specification is simply taken from a library provided by the system and mapped to the token name for the particular problem instance.

The same principle can be applied for other subtasks which have characteristics that frequently occur in the problem domain. Eli provides a library of reusable specification modules for different instances of name analysis tasks. They are used by specifying that certain symbols of the particular tree grammar play computational roles relevant for name analysis. In Figure 3 the module for Algol-60-like scope rules is chosen. The roles of the grammar root, the range of definitions, defining and applied identifier uses are mapped to the grammar phrases.

```
SYMBOL Progr  INHERITS Root  END;
SYMBOL Block  INHERITS Range END;
SYMBOL VarDef INHERITS IdDef END;
SYMBOL VarUse INHERITS IdUse END;
```

Fig. 3. Use of a name analysis specification module

Such specification modules encapsulate precoined descriptions of certain common problem instances. They are formulated in terms of the specification languages for their subtasks, e. g. regular expressions for tokens, attribute grammars for name analysis. The user maps the central concepts of the identified problem to entities of his specification without knowing the details of the specification module.

This method introduces a powerful higher level of specification. Of course its application is restricted to a subset of problem instances within the problem domain. That range can be rather large if there are many such precoined specifications, and if they can be flexibly combined to achieve individual solutions. Those are general criteria for reusable libraries.

Specification modules like that applied in Figure 3 for the consistent renaming task introduce a new kind of declarative specifications. Such a module describes a set of related task specific concepts, in this case **Root**, **Range**, **IdDef**, and **IdUse**. In case of Algol-60-like scope rules a **Range** is a program phrase that has a mapping from identifier encodings to object keys describing identifier bindings within that phrase. For nested ranges that mapping is obtained by the hiding rule. **Root** is a phrase that contains all ranges of a program. **IdDef** contributes an identifier-key binding for the smallest enclosing range. **IdUse** yields the key bound to that identifier in the smallest enclosing range. A specification like that in Figure 3 states which phrases of the particular tree grammar play the roles of the related concepts described by the module. (Usually **Range**, **IdDef**, and **IdUse** are mapped to several grammar symbols).

On this level of abstraction it is rather easy to show that the specification describes the intended properties, provided the module itself is specified correctly.

The module is described in terms of dependent computations using the calculus of attribute grammars. Figure 4 shows the module specification for our example taken from [6]. The verification of the above described concepts is supported by concentration on a single computational aspect (here consistent renaming), and by abstraction from the particular tree structure. For example it can easily be shown that the **KeyInEnv** function in the **IdUse** context is not called before all **DefineIdn** calls are done for all **IdDef** contexts in enclosing ranges. On the level of attribute grammars such dependencies between computations, and the propagation of values between computations can be proven.

```
SYMBOL Range: Env: Environment, GotLocalKeys, GotAllKeys: VOID;
SYMBOL IdDef, IdUse: Sym: int, Key: DefTableKey;

SYMBOL Root INHERITS Range COMPUTE
  INH.Env = NewEnv();
  INH.GotAllKeys = THIS.GotLocalKeys;
END;

SYMBOL Range COMPUTE
  INH.Env = NewScope(INCLUDING Range.Env);
  INH.GotAllKeys = THIS.GotLocalKeys
    DEPENDS_ON INCLUDING Range.GotAllKeys;
  SYNT.GotLocalKeys = CONSTITUENTS IdDef.Key;
END;

SYMBOL IdDef COMPUTE
  SYNT.Key = DefineIdn(INCLUDING Range.Env, THIS.Sym);
END;

SYMBOL IdUse COMPUTE
  SYNT.Key = KeyInEnv(INCLUDING Range.Env, THIS.Sym)
    DEPENDS_ON INCLUDING Range.GotAllKeys;
END;
```

Fig. 4. An Attribution Module for ALGOL 60-like Scope Rules

The operational aspects, i. e. the effects of the computations, are beyond the scope of this calculus. In [5] a formal specification of an abstract data type for the functions used in this module is given. An implementation can be proven against that specification, but it cannot be used to generate the implementation.

This observation applies as well to the other kinds of declarative specifications mentioned above (scanner, parser, tree construction, output patterns): The declarative aspects can be proven within the corresponding formal calculus. The correctness of their implementation is preserved by the generating tools. For the operational hooks the sequence of calls and the supply of arguments can be

proven within the calculus, tools can integrate the operations correctly into the generate algorithm, but the effect of those operations can not be proven in the original formal model.

5 Conclusion

We have shown that executable specifications can be achieved in the domain of language implementation. Narrowing the problem space to a certain application domain leads to specification strategies that effectively simplify specification development and support generation of high quality implementations:

A domain specific decomposition model supports well-structured specifications using dedicated formal calculi for subproblems. Dedicated tools can be applied to components of the specifications. Careful distinction between declarative formal specifications and necessary operational hooks allows tools to perform effectively on exactly their task.

Most implementation decisions can be made without explicit specification by an integrated system that embodies know-how of domain specific experts.

In a restricted problem domain there are usually widely applicable and well-understood common concepts. They give rise to specifications on higher levels, and to the use of precoined solutions.

It seems to be promising to transfer these strategies and the experience of this area to other applications domains.

Acknowledgements The ideas and insights presented here are a result of the cooperation with W. M. Waite on the development of the Eli system over many years. This work was partially supported by the government of Nordrhein-Westfalen through the SofTec-Cooperation.

References

1. Cleaveland, J. C.: Building Application Generators. IEEE Software **5** (July 1988), 25–33
2. Gray, R. W., Heuring, V. P., Levi, S. P., Sloane, A. M. & Waite, W. M.: Eli: A Complete, Flexible Compiler Construction System. Communications of the ACM **35** (Feb. 1992), 121–131.
3. Horowitz, E., Kemper, A. & Narisimhan, B.: A Survey of Application Generators. IEEE Software **2** (Jan. 1985), 40–54.
4. Kastens, U.: Attribute Grammars in a Compiler Construction Environment. Proceedings of the International Summer School on Attribute Grammars, Application and Systems, Lect. Notes in Comp. Sci.# **545**, Springer Verlag, New York-Heidelberg-Berlin, 1991, 380–400.
5. Kastens, U. & Waite, W. M.: An Abstract Data Type for Name Analysis. Acta Informatica **28** (1991), 539–558.
6. Kastens, U. & Waite, W. M.: Modularity and Reusability in Attribute Grammars. Universit t-GH Paderborn, Reihe Informatik, July 1992.

7. Krueger, C.W.: Software Reuse. ACM Computing Surveys **24** (June 1992), 131–183.
8. Spivey, J. M.: The Z Notation A Reference Manual, 2nd Ed. International Seriers in Computer Science, Prentice Hall, 1992.
9. Waite, W. M.: A Complete Specification of a Simple Compiler. Department of Computer Science, University of Colorado, Boulder, CU-CS-638-93, Jan. 1993.