Expressing Structural Properties as Language Constructs^{*}

Shriram Krishnamurthi, Yan-David Erlich, and Matthias Felleisen

Department of Computer Science
Rice University
Houston, Texas 77005-1892, USA

Abstract. A program should document its organization and decisions about the programming process. Since the programmer's thinking about programming and program organization continually evolves, languages inevitably prove unable to state these decisions in a precise and adequate fashion. Macro systems could provide a convenient way to extend a language with such statements, if they had more structure than traditional C- and Lisp-style macros provide.

With our system, McMicMac, designers can express a variety of specifications as language constructs, including program representations of design patterns, high-level recursive programming operators, and collaboration-based design mechanisms. Unlike traditional macro systems, McMicMac offers a simple yet powerful means for describing specifications, prevents unintentional name clashes, provides feedback in terms of the programmer's source, and has modular mechanisms for managing specifications. We have implemented and used McMicMac to define several groups of extensions.

1 Introduction

A program specifies not only behavior, but also decisions concerning its organization and about the programming process. It expresses this information in terms of the constructs of the programming language in which it is written. Any language, however, has only a limited set of constructs, while programmers constantly invent new concepts and grow their vocabulary about both the program and the programming process. This leads to a gap in expressiveness.

Many programming methodologies try to narrow this gap. Program design patterns [12] that capture some details of the structure of the program are one example; designs based on collaborations [3] are another. Yet other examples are abstract structure operators [22] and adaptive programming specifications [20]. They permit programmers to describe a program's organization and development, and can also be mapped into constructs in the underlying programming language.

^{*} This research was partially supported by NSF grants CCR-9619756 and CCR-9708957, and by a Texas ATP grant.

These specifications are used in various ways. They are typically described only in comments, translated into code by hand, or embedded in the program as linguistic extensions that are processed by special-purpose tools. Comments are unchecked and therefore unreliable. Implementing specifications manually is painstaking, deprives program maintainers of crucial information, and makes it difficult to ensure that constantly-evolving programs are meeting the intended specifications. In contrast to these, special-purpose tools can both verify that the program satisfies the specification and translate the specifications into instructions in the underlying programming language.

Unfortunately, these special-purpose tools take considerable effort to implement. Each tool must parse the complete programming language and its own extensions, and implement its own rewriting strategy. Also, since these tools hard-wire their understanding of the underlying language, they may not be able to process a program extended with constructs from some other specification process. Finally, extending such tools may be cumbersome, and can differ considerably between tools.

We propose an alternate approach to expressing such structural properties. Our approach, based loosely on Scheme macros, offers more structure and more facilities than traditional C or Lisp-style macro expanders. It simplifies the task of specifying properties, and tries to preserve their integrity when communicating information between programming tools and the programmer. Finally, by providing a common framework, it allows different specifications to interact atop the same language and with each other.

The rest of this paper is organized as follows. Section 2 discusses some of the structural properties that programmers use, and how they can be implemented. Section 3 describes features that would be useful to implement the specifications of Section 2, and how our implementation supports them. Section 4 surveys the literature of related work. The last section summarizes the ideas in this paper, and presents some directions for future work.

2 Programming with Specifications

In this section we present some examples of structural properties and discuss how to integrate them into the programming process.

2.1 Software Patterns

Design patterns sketch a solution to a class of related problems. By customizing a pattern to a given context, a programmer can reuse the embodied solution. Some patterns represent general “architectures” with respect to a programming language that can be instantiated effectively in different contexts. We call these architectures *software patterns*.

Figure 1 illustrates a sample use of a software pattern¹ in Java. The code in the figure uses the Adapter pattern, which enables the reuse of an existing

¹ All the patterns mentioned in this paper are taken from Gamma, et al. [12].

```
interface OutputRoutines {
    void drawLine ();
    void printText (); }
class VendorGraphics {
    void drawSolidLine () {};
    void renderText () {}; }
// GraphicalOutput adapts VendorGraphics to OutputRoutines
class GraphicalOutput implements OutputRoutines {
    VendorGraphics lowLevelDriver;
    public void drawLine () { lowLevelDriver.drawSolidLine (); }
    public void printText () { lowLevelDriver.renderText (); } }
```

Fig. 1. Adapter Pattern: Java Version

```
(interface OutputRoutines
  (methods (void drawLine ())
            (void printText ())))
(class VendorGraphics
  (methods (void drawSolidLine ())
            (void renderText ())))
(Adapter GraphicalOutput adapts VendorGraphics
      to OutputRoutines
      as lowLevelDriver

(fields)
(methods
  (public void drawLine () (lowLevelDriver . drawSolidLine ()))
  (public void printText () (lowLevelDriver . renderText ())))
```

Fig. 2. Adapter Pattern: Specification-Based Version

class with a different interface. More specifically, suppose an existing class implements the desired functionality but does not implement the desired interface. An Adapter acts as a surrogate for this class by implementing the desired interface, forwarding requests to instances of the existing class and tailoring responses to its interface.

In our example, *OutputRoutines* is an interface that represents output generators, and is implemented by actual generators on various devices. The vendor-provided *VendorGraphics* performs graphical output, but *VendorGraphics* does not implement *OutputRoutines*, and its interface is slightly different. In Figure 1, the programmer creates an Adapter class, *GraphicalOutput*, that forwards requests to an instance of *VendorGraphics*. (For brevity, we elide the details of the actual methods.) The relationship is documented informally through the comment.

The programmer could now make the mistake of defining the class *Client* with the type of some variable *var* to be *VendorGraphics* but assigned an instance

of *GraphicalOutput*. Unfortunately, *VendorGraphics* is the class being adapted rather than the interface it is being adapted to:

```
class Client {
    VendorGraphics var = new GraphicalOutput ();
}
```

The type-checker flags this assignment with the following error message:

```
test.java:13: Incompatible type for new. Can't convert
    GraphicalOutput to VendorGraphics.
    VendorGraphics var = new GraphicalOutput ();
                           ^
```

This example illustrates several effects of programming directly in terms of a pattern's constituent constructs:

- The implementation of the Adapter pattern in terms of plain classes and interfaces obscures the pattern's identity, and thus decreases the clarity of the program. This makes it more difficult for readers to understand the structure and intent of the code.
- It increases the potential for errors due to the volume of code. Pattern code handles administrative tasks such as maintaining invariants, which a programmer must correctly implement during the initial development and remember to update during maintenance. Even in this simple example, for instance, the programmer must remember to make *GraphicalOutput* implement the interface *OutputRoutines*, and declare the type of *lowLevelDriver* as *VendorGraphics*.
- It becomes difficult to replace the code for the pattern instance with an improved version, e.g., one that is more extensible or offers better protection against errors.
- All feedback is reported in terms of the individual units of code, and the programmer must then manually extrapolate from the error message to the original pattern-based design.

In short, programming directly with the constituents of patterns can potentially affect programmers at every stage: implementation, debugging and maintenance.

Figure 2 shows the same program (translated into a parenthesized version of Java), except it makes the Adapter explicit by using an **Adapter** construct, which is translated into the equivalent of the code in Figure 1. The use of an Adapter makes it clear that there is a mismatch between *OutputRoutines* and *VendorGraphics*, and highlights how this can be overcome. The equivalent of *Client*'s Java declaration is then

```
(class Client
    (fields (VendorGraphics var = (new GraphicalOutput ())))
```

This results in the error message

```
test.cj:15.33-15.56: Incompatible assignment type. Can't convert
  GraphicalOutput to VendorGraphics.
GraphicalOutput created by Adapter at test.cj:7.2-7.8.
```

which uses the Adapter declaration to report the error in terms of the programmer's pattern code, not just the expanded constructs. The error message frees the programmer from having to track manually what each construct can introduce, which can be especially difficult since the expansion of the construct can change over time. This error message is produced by a generic type-checker for our parenthesized version of Java; Section 3.3 describes how this information is generated in greater detail.

2.2 Composition Invariants

Smaragdakis and Batory [24] present several approaches to validating the correctness of module compositions. They express the compositional requirements through the type system. For instance, suppose the implementation of a virtual machine could include a module that tags the memory representations of data (*Tag*) and another that implements garbage collection (*GC*). Furthermore, suppose that the *GC* module requires *Tag* to be present in the module composition. To ensure that the *Tag* module has already been added, *GC* declares a dummy variable of type *TagIncludedProperty*, which is the name of an empty class declared only by the *Tag* module. Thus, if *Tag* is not included in the composition hierarchy before *GC*, the dummy variable declaration will raise a type error, which indicates an erroneous composition.

Implementing this approach via explicit constructs for validating compositions offers many advantages:

1. These class and variable declarations play no role in the execution of the program. Therefore, they obscure its behavior, potentially making it more difficult to maintain.
2. Some of these invariants are tricky to implement. It is thus safer to have them implemented automatically by a program than manually by the programmer. This is especially important since some negative properties require a quadratic number (in the size of the program and the number of properties) of dummy variable declarations. Also, when a new property is introduced, the programmer has to add it to each class by hand. With a properly designed language extension, the programmer can instead establish a single point of control and greatly reduce the work required to specify the program's properties.
3. When an invariant is violated, the programmer should get an error in terms of the actual invariant, not just in terms of the (often mangled) names used to represent it in the source. Smaragdakis and Batory identify this issue as one of the major problems in their approach [24, pg. 562].
4. While adding the dummy variable declarations, a programmer might accidentally choose a name that is already in use for a different purpose.

2.3 Other Specifications

Several other specification techniques can be handled with our proposed solution. For instance, abstract structure operators [22] are used to simplify the process of describing traversals over recursively-defined types. They are generated automatically from the type specification. Instead of defining recursive procedures over these types, programmers use and compose these high-level operators, which are then translated into regular recursive procedures. The translation again raises the issues of name-clash management and feedback-reporting, which can be handled by our proposal.

3 Programming Support for Specifications

In the preceding section, we have mentioned several desirable properties for a programming tool that supports specifications. To summarize, such a tool

1. should make it easy to describe the syntax of the property;
2. should permit a simple specification of the translation into code fragments for those properties that require an implementation;
3. must help the designer avoid name capture problems;
4. should provide feedback to programmers in terms of the original specifications, not just their expansions; and,
5. should support the definition of disjoint groups of specifications.

We have developed a language elaborator, McMicMac, that meets these requirements. In this section, we describe the key properties of McMicMac, and explain how they address the aforementioned needs. For information concerning McMicMac's implementation, we refer the reader to the technical report [18].

3.1 Describing Specifications

Many specifications have operational meanings. For such specifications, *designers must be able to specify the transformation that maps an instance of a specification into constructs in the underlying language*. The mechanism must be both convenient enough to simplify the addition of new transformations, and powerful enough to specify potentially complex ones.

A software pattern, for example, can be thought of as a parametric body of code that is specialized at each particular use. To use a pattern, the programmer must provide code fragments for the parameters. The pattern implementation assembles a program component by splicing the code fragments into the parameter positions of the body of code. Thus the pattern designer needs a tool that (1) permits the definition of parameterized bodies of code, and (2) generates code from arguments supplied for the parameters. McMicMac supports these operations felicitously through shape matching,² a mechanism due to Kohlbecker and Wand [17].

² This is traditionally known as “pattern matching”, but we adopt different terminology to avoid confusion with software patterns.

```
(define-macro Adapter
  (rewrite
    (Adapter adapterName adapts adapteeType
              to desiredInterface
              as adapteeVariable
              (fields field-decls ...)
              (methods method-decls ...))
    (with-keywords adapts to as fields methods)
    (as
      (class adapterName implements desiredInterface
        (fields (adapteeType adapteeVariable)
                 field-decls ...)
        (methods method-decls ...))))))
```

Fig. 3. Adapter Pattern Definition

Shapes are built from keywords, shape variables, and sequences. The shape-matcher works by comparing each phrase in a program against the collection of defined shapes. A phrase matches a shape if the phrase uses keywords and sequences in the same manner as the shape. In this case, the shape matcher binds the corresponding parts of the inputs to the shape variables in the template. It then generates an output phrase based on the corresponding macro definition, which yields a new phrase. It expands this phrase again, continuing until the phrase does not match any defined shape. Figure 3 presents the definition of the Adapter pattern, which is used in Figure 2. This use results in code that is equivalent to that of Figure 1.

The matcher treats ellipses (...) specially. An ellipsis follows a (“head”) shape in a sequence,³ and matches a source sequence of zero or more instances of the head shape. It binds each shape variable in the head shape to a sequence. This sequence consists of the sub-terms, in order, of the terms in the source sequence that correspond to the shape variable’s position in the head shape. Ellipses can be nested to arbitrary depth. Each nesting level introduces a nested sequence in the binding of a shape variable.

The definition and application of the Adapter pattern illustrate the use of ellipses. One sub-shape of the Adapter is (**methods** *method-decls* ...). In the example, there are two methods in the shape:

```
(methods
  (public void drawLine () (lowLevelDriver . drawSolidLine ()))
  (public void printText () (lowLevelDriver . renderText ())))
```

The shape-matcher binds *body-terms* to a sequence of two pieces of code:

```
(public void drawLine () (lowLevelDriver . drawSolidLine ()))
```

³ The head of a shape cannot be ..., else the shape is considered ill-formed.

and

```
(public void printText () (lowLevelDriver . renderText ()))
```

As Figure 3 suggests, the shape-matcher not only verifies that the specification has a proper form, it also generates output. Put differently, adding a specification via *McMicMac* assigns a precise meaning to a specification. The shape-generator works by essentially inverting the matching process. In particular, it too can handle nested ellipses; each shape variable must appear under as many ellipses in the output shape template as it did in the input template.

Ellipses play an important role not only in laying out the notation, but also in defining the meaning of specifications. They elucidate the inductive structure of the specifications, an aspect that is often overlooked in pattern definitions. Specification implementors use ellipses wherever they want to leave the number of concrete entities unspecified and let the programmer provide this information. In the Visitor pattern, for instance, the programmer’s specification not only generates the visited type but also the visiting methods that characterize the pattern. Thus ellipses capture what Lauder and Kent term “purity” [19]: they reflect *all* instances of the pattern, not just a single deployment from which the programmer must extrapolate to the general case.

3.2 Avoiding Inadvertent Interference in Code

When a specification expands into code that co-mingles with code written by the programmer, one code fragment might inadvertently bind or use a name that is bound or used in the other fragment. Since neither the programmer nor specification designer can foresee all such situations, *the elaborator must ensure that the programmer’s and the specification’s code do not inadvertently interfere with the lexical properties of each other.*

Macro expansion can ensure that variable bindings and uses in user code do not bind and are not accidentally bound by bindings and uses in generated code, and vice versa. This process is called *hygienic* expansion [16]. In *McMicMac*, the hygienic macro expander works by marking each step of expansion with a distinct “timestamp”, and accumulating timestamps on every term. To determine whether a variable is bound, the expander considers both the name and timestamps on a variable, and identifies only those variables with the same name and timestamps. Since code from the user and from each macro elaboration have different timestamps, this prevents inadvertent capture of variables.

For example, consider Smaragdakis and Batory’s method for verifying modular compositions [24]. Their specification mechanism introduces declarations of the following form (in C++, using our example from Section 2.2):

```
template < class Super > class Tag : public Super {
protected:
    class TagIncludedProperty {};
    ... }
```


is the *Tag* module, and

```
template < class Super > class GC : public Super {
private:
    TagIncludedProperty dummy1;
    ... }
```

is *GC*. The variable *dummy1* here has two potential problems:

- It interferes with any other attempt to declare a variable of the same name. Such a variable may be declared not only by the user, but also by some other specification mechanism. Even if the specification designer chooses an obscure name, some specifications can be applied multiple times, so one instance of the specification can clash with another one.
- Some other part of the class's body may accidentally mention the name *dummy1*. This should produce an unbound variable error, but because of the above declaration, it may produce an unexpected type error or, worse, run without error and compute an incorrect answer.

Hygienic expansion ensures that the identifier *dummy1* that is introduced by the macro does not conflict with any other identifiers, including those named *dummy1* but coming from other sources (including other uses of the same macro). This is essential when a program is developed by a team of programmers, who would otherwise find it extremely difficult to prevent inadvertent name collisions.

3.3 Maintaining the Integrity of Specifications in Feedback

A programming environment includes numerous tools that provide feedback about the program. These include type-checkers, program slicers, debuggers, profilers, and so forth. In all these cases, the analyzed program includes both code written by the programmer and code generated by a specification. Even when these are co-mingled, *feedback must be in terms of the source written by the programmer, not just in terms of the generated code*. This is essential for two important reasons:

- to maintain the impression that the programmer is coding at the level of the specification, not at the level of the code it generates; and,
- because some of the code in question may not appear in the source text.

Feedback of this form is particularly important when the juxtaposition of user and specification code results in complex errors.

McMicMac provides an interface for tools that wish to report feedback. The interface consists of two kinds of information, and protocols that process this information. The two kinds of information are:

source-correlation For each term, McMicMac notes the source location of the phrase that generated it. This information is maintained through the macro expansion process. Thus when a tool needs to provide feedback about a term,

it can report it in terms of the source phrase that the programmer needs to examine. This is especially helpful to graphical interfaces, since they can highlight the appropriate source text.

elaboration-tracking McMicMac also maintains a history of transformations applied to each elaborated term. In the example of Section 2.1, it records that *GraphicalOutput* is generated by **Adapter** (with its source location). In more complex examples, one specification might expand into the use of another. In these cases, conventional error messages in terms of the constituents might be especially unhelpful.

The tools themselves do not need to process this information. For example, the McMicMac error checker processes only the CLASSICJAVA language, and has no knowledge of software patterns. Instead, the information is extracted and processed by a McMicMac protocol, which generates feedback of the sort shown in Section 2.1.

3.4 Organizing Specifications by Layers

Specification designers need modular constructs to organize collections of specifications. McMicMac provides a module mechanism called *vocabularies*. Each specification must be placed in some vocabulary; a vocabulary represents a language fragment, derived by combining the individual macros in that vocabulary. Vocabularies can be combined with other vocabularies. In particular, a vocabulary representing a specification can be combined with different base languages. These base languages might vary in at least two different ways:

1. By layering several vocabularies atop a base language, the programmer can combine multiple specification schemes. This may be impossible to achieve when each specification method provides its own tool, since these tools may reject or incorrectly process the other forms of specification.
2. Users can customize how they use the specification method by picking different base languages. For instance, a sophisticated programmer will want a powerful base language, while teachers might prefer a simpler one for their students.

A vocabulary also represents one way in which a specification analyzes a program; some specifications may process the same program fragment several times. For instance, many common program analyses and optimizations can be partitioned into “collection” and “modification” phases. The first phase traverses the program and collects information relevant to the analysis. This information is processed, and then used by the second phase, which rewrites the program appropriately. Each of these phases can be elegantly encoded as a vocabulary.

3.5 Implementation Status

McMicMac is currently implemented in MzScheme [9]. We have constructed vocabularies for several versions of Scheme [8] and one for CLASSICJAVA [10], an

idealized, parenthesized subset of Java. We have enhanced these base languages with several structural transformers, including software patterns corresponding to each of the design patterns described in the catalog of Gamma, et al. [12], abstract structure operators [22], and execution profilers [1].

The facilities of McMicMac helped us implement all these tools in short order. Our implementation produces information that helps us generate useful and intuitive feedback for the programmer, such as the error messages shown in this paper. Though the current implementation of McMicMac is restricted to parenthesized syntax, it can be extended to more traditional syntaxes along the lines of Cardelli, et al.'s work [6].

4 Related Work

Researchers have studied various specification mechanisms. We partition these into several categories below, though naturally there is some overlap.

4.1 Software Patterns

With the increasing popularity of software patterns, many researchers have proposed ways to design tools and languages that support them.

Graphical and Translation Tools. Patterns are frequently described using graphical notations. Researchers have thus tried to design graphical tools that support programming with patterns. Budinsky, et al. [5] describe a GUI-based package which lets users prepare a diagrammatic representation of the pattern-level layout of their program, and then fill the components with source code. The system's back-end uses a script to generate code from the user's input. Florijn, et al. [11] have built browsers to support their work on fragments, described below. Meijler, et al. [21] have built a modeling environment called FACE in which programmers use patterns by cutting-and-pasting from a library of patterns in UML, and then filling in the blanks with code. The FACE environment is unusual in that it supports patterns throughout the program's life, not just during its initial creation.

Graphical tools are attractive because they can provide an intuitive interface to the programmer, but they have several shortcomings.

1. They often do not clarify the distinction between a pattern, which can have a potentially unbounded number of components, and a pattern instance, which has a fixed, finite number. Often, they provide a template with a small number of placeholders for concrete entities, which the user must duplicate by hand to obtain the desired number; this process is both laborious and error-prone.
2. Some contain an analysis engine for the pattern language, but none of them interact with the underlying programming language to provide an equally convenient interface for feedback once the patterns have been translated into code. This could be remedied by using a system such as ours as a back-end for the graphical tools.

3. The systems descriptions do not clarify how groups of patterns can be organized, or how to extend tools to accommodate other specification notations.

Soukup [25] proposes implementing patterns through C-style macros. These macros lack inductive constructs, offer no facilities for reporting feedback, are not hygienic, and have no modular structure. Hedin [13] defines patterns in terms of attribute grammars. His work currently uses these to reconstruct patterns in existing source code.

Constraint Models. Both Bosch [4] and Florijn, et al. [11] describe interesting approaches to pattern-based programming. Bosch takes a top-down view on programming with patterns. His system uses a layered object model to describe constraints on entities that participate in patterns. His paper also lays out four significant problems in implementing design patterns, of which our work addresses three: *traceability*, *reusability* and *implementation overhead*.

Programmers who use Florijn, et al.'s system write "fragments", which are elements of programs and patterns. They then designate fragments to play roles in instances of patterns. The pattern designer specifies constraints (similar to contracts [14]) that judge when a collection of fragments satisfies the pattern. This approach gives the programmer the flexibility of restructuring the system easily. It is, however, unclear how their system handles feedback across elaboration or the inductive structure of patterns.

4.2 Other Specification Implementations

Adaptive programs [20] consist of declarations of class hierarchies, traversals, and visitors. The traversal maps the order in which to visit nodes in the hierarchy, while the visitors determine what actions to perform at each node. Adaptive programs are compiled into programs in a conventional language like Java using the Demeter tool [20]. Though it uses sophisticated algorithms to compile specifications into efficient code, Demeter does not report feedback across the elaboration process.

Aspect-Oriented Programming [15] is implemented by *weavers*, which combine the original program with code derived from the aspect specifications. The implementation for Java, called AspectJ, produces generic Java code that can be compiled by any generic compiler. As a result, however, it does not offer the user support at the level of aspect specifications. Instead, the primer [27] instructs users to track errors based on their file location.

Batory and Geraci [2] describe a process for checking that code generated from specifications meets compositional requirements. Their work imposes a type system on a program's components, and uses this system to verify the validity of compositions. When a composition is invalid, the system can suggest reasons for the error. Their work can be used to improve the error reporting for modular, collaborative designs [24].

4.3 Macro Systems and Extensible Grammars

McMicMac is a close relative (and descendent) of other language extension mechanisms, including macro systems and extensible grammars. The derivation and use of McMicMac’s shape-matching mechanism was described by Kohlbecker and Wand [17]. The design of hygienic macro expanders is due to Kohlbecker, et al. [16], though our algorithm is derived from that of Dybvig, et al. [7]. The latter work also outlines a method for maintaining source-correlation.

Like a macro system, Taha and Sheard’s MetaML [26] gives the user the ability to manipulate fragments of a program’s source as values. Their system performs type-checking to ensure that program compositions do not produce type errors, and verifies that phases of elaboration do not interfere with each other in ill-defined ways. Abstract structure operators [22], described in Section 2.3, were implemented using Compile-time Reflective ML (CRML) [23], a predecessor of MetaML.

Cardelli, et al. [6] describe a system that is similar to modern macro systems but admits more flexible syntaxes. Their approach is based on extensible grammars and does not incorporate a direct equivalent of our ellipses. They impose a type discipline on terms, which they use to ensure that elaboration terminates. Though their system allows users to extend the grammar of a language, the extensions are not defined in modular blocks (such as McMicMac’s vocabularies) that can be combined with different base languages.

5 Conclusion and Future Work

We have shown how modern macros can be used to extend programming languages with constructs for specification mechanisms. Modern macro systems offer many features that make such extensions both convenient and powerful. First, macros support a simple and powerful shape-matching notation to define specifications. Second, macros automatically prevent insidious arrangements of code from affecting the lexical properties of both specification and user code. Third, macros can be grouped into modular constructs that make the macros more customizable and reusable. Most importantly, macros can track source locations and elaboration history and thus present users with helpful feedback, e.g., types, data flow and other messages, in terms of their source. This last issue, which is crucial for software development, is frequently ignored in the literature on software tools.

We anticipate many future directions for this work. First, our framework makes it easy to add program-processing tools. We can build tools that analyze, verify and document programs by selectively processing specifications that they understand, and falling back on the standard expansions for unfamiliar ones. Second, we expect the framework can accommodate other paradigms like adaptive programming [20]. One vocabulary could collect the adaptive specifications and program structure, while a second one would rewrite the program, removing the specifications and inserting code to perform traversals. Finally, by implement-

ing numerous platforms in this framework, we can study the properties that are common to structural specifications.

Acknowledgments We thank the anonymous referees for their detailed remarks.

References

1. Appel, A. W., B. F. Duba, D. B. MacQueen and A. P. Tolmach. Profiling in the presence of optimization and garbage collection. Technical Report CS-TR-197-88, Princeton University, November 1988.
2. Batory, D. and B. J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, pages 67–82, February 1997.
3. Beck, K. and W. Cunningham. A laboratory for teaching object-oriented thinking. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 1–6, 1989.
4. Bosch, J. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 1998.
5. Budinsky, F. J., M. A. Finnie, J. M. Vlissides and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2), 1996.
6. Cardelli, L., F. Matthes and M. Abadi. Extensible syntax with lexical scoping. Research Report 121, Digital SRC, 1994.
7. Dybvig, R. K., R. Hieb and C. Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, December 1993.
8. Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs*, pages 369–388, 1997.
9. Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.
10. Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Symposium on Principles of Programming Languages*, pages 171–183, January 1998.
11. Florijn, G., M. Meijers and P. van Winsen. Tool support for object-oriented patterns. In *European Conference on Object-Oriented Programming*, pages 472–495, 1997.
12. Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Personal Computing Series. Addison-Wesley, Reading, MA, 1995.
13. Hedin, G. Language support for design patterns using attribute extension. In Bosch, J. and S. Mitchell, editors, *European Conference on Object-Oriented Programming Workshop Reader*, pages 137–140, 1997.
14. Helm, R., I. M. Holland and D. Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, pages 169–180, 1990.
15. Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier and J. Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, June 1997.

16. Kohlbecker, E. E., D. P. Friedman, M. Felleisen and B. F. Duba. Hygienic macro expansion. In *ACM Symposium on Lisp and Functional Programming*, pages 151–161, 1986.
17. Kohlbecker, E. E. and M. Wand. Macros-by-example: Deriving syntactic transformations from their specifications. In *Symposium on Principles of Programming Languages*, pages 77–84, 1987.
18. Krishnamurthi, S. PLT McMicMac: Elaborator manual. Technical Report 99-334, Rice University, Houston, TX, USA, 1999.
19. Lauder, A. and S. Kent. Precise visual specification of design patterns. In *European Conference on Object-Oriented Programming*, pages 114–134, July 1998.
20. Lieberherr, K. J. *Adaptive Object-Oriented Programming*. PWS Publishing, Boston, MA, USA, 1996.
21. Meijler, T. D., S. Demeyer and R. Engel. Making design patterns explicit in FACE, a framework for adaptive composition environment. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 94–110, September 1997.
22. Sheard, T. Automatic generation and use of abstract structure operators. *ACM Transactions on Programming Languages and Systems*, 13(4):531–557, October 1991.
23. Sheard, T. Guide to using CRML: Compile-time Reflective ML. Technical report, Department of Computer Science and Engineering, Oregon Graduate Institute, November 1992.
24. Smaragdakis, Y. and D. Batory. Implementing layered designs and mixin layers. In *European Conference on Object-Oriented Programming*, pages 550–570, July 1998.
25. Soukup, J. Implementing patterns. In *Pattern Languages of Program Design*, pages 395–412, 1995.
26. Taha, W. and T. Sheard. Multi-stage programming with explicit annotations. In *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217, 1997.
27. Xerox PARC AspectJ Team. The AspectJ primer. Web document <http://www.parc.xerox.com/spl/projects/aop/aspectj/>, January 1999.