© ACM, (2001). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in SSR '01: Proceedings of the 2001 symposium on Software reusability, http://doi.acm.org/10.1145/375212.375258

Generation of visitor components that implement program transformations

R. E. Kurt Stirewalt,* Laura K. Dillon†
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824
{stire,ldillon}@cse.msu.edu

ABSTRACT

The visitor pattern is appealing to developers of program-analysis tools because it separates the design of the data structures that represent a program from the design of software that traverses these structures. Unfortunately, the visitor pattern is difficult to apply when the analysis involves transformation logic that involves multiple program fragments simultaneously. We encountered this problem in our work on the Amalia project and discovered a novel way to use multiple cooperating visitor objects to systematically implement such functions when they are specified via a set of transformation rules. This paper introduces our *curried-visitor framework* and illustrates how we applied it to implement a key component in the Amalia framework. We are working on a code generator that will automatically synthesize curried-visitor frameworks from a description of a program's abstract syntax and a set of pattern-matching transformation rules.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques; D.1.2 [**Programming Techniques**]: Automatic Programming—*Program synthesis, Program transformation*; D.2 [**Software**]: Software Engineering; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented Programming*

General Terms

Design, Languages

Keywords

Amalia, Curried visitor framework, Lightweight analysis components, Program transformations, Visitor pattern

*Partial support provided by NSF grants CCR-9901017 and EIA-0000433

†Partial support provided by NSF grant CCR-9896190 and EIA-0000433

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SSR'01, May 18-20, 2001, Toronto, Ontario, Canada.
Copyright 2001 ACM 1-58113-358-8/01/0005 ...$5.00.

1. INTRODUCTION

The visitor pattern is appealing to developers because it decouples the design of complex linked object structures from the design of functions that traverse these structures [9]. Visitors are particularly attractive in program-analysis and CASE applications, where programs are represented as *abstract syntax trees* (ASTs), which are linked structures of objects whose classes correspond to types of statements and expressions in the programming language. Many a useful program analysis can be specified as a function that maps the ASTs of programs into a domain of facts about these programs. We recently developed the Amalia framework for automatically generating a visitor-based analyzer from a declarative specification of such a function [20, 8]. Since developing Amalia, we have discovered many interesting analysis functions that are amenable to a visitor-based implementation, but that are not expressible in the Amalia specification language. This paper extends our work in the automatic generation of visitor-based program analyzers to a larger class of analyses.

Generally speaking, the visitor pattern accommodates algorithms that: (1) traverse a composite object structure depth-first, (2) dispatch functions on objects in the structure as the objects are visited, based on their class, and (3) exploit object-orientation to carry information either up (toward the root) or down (toward the leaves) during the traversal. This second property—dispatching a function on an object based on the class of the object—illustrates that the operation implemented by a given visitor is *polymorphic* over its argument. Many interesting analyses (e.g., next-step analysis of process-algebraic languages) can be formalized as operations that are polymorphic over a single (AST) parameter. Amalia automatically generates visitors that implement such operations. There are also many interesting analyses that are formalized as operations that are polymorphic over more than one parameter. For example, an algebraic simplifier for an expression language dispatches a simplification function based on the class of two (or more) expression operands. As another example, in Amalia, we had to implement a *normal-form reduction* operation that applies to ASTs of formulas in linear temporal logic (ltl). Unfortunately, it is not immediately clear how to apply the textbook visitor pattern to implement operations that are polymorphic in more than one parameter.

This paper describes a general pattern for designing visitor-based implementations of operations that are polymorphic in more than one structured-data parameter. We call our solution a *curried visitor framework* because its structure and operation are reminiscent of currying in functional programming languages, such as ML and Haskell. A curried visitor framework efficiently implements an n-ary polymorphic function by exploiting multiple visitor classes, the instances of which cooperate at run-time to compute the n-ary func-

tion. Intuitively, a curried visitor framework implements a polymorphic function over n arguments with a visitor object that traverses the first argument in order to *choose* a visitor object that implements a polymorphic function over the remaining $n-1$ arguments. In the case where $n = 1$, the visitor object is just a simple visitor, i.e., the kind described in the patterns literature and generated by Amalia.

Curried visitor frameworks cannot be generated using the visitor-generation technology we developed in [20, 8]. On the other hand, we argue that they can be generated from a set of pattern-matching transformation rules. We applied curried visitors to design the normal form reduction of temporal logic formulas. This construction combines sub-formulas of two formulas that are already in the canonical form to construct a new composite formula. In this case, two formulas are being combined, and the rules for combining them are a function of both the types of the formulas and the type of the combining operator. We specified this construction as a set of transformation rules. The curried visitor framework provides a different function scope for each distinct pattern in our rules. With these distinct scopes available, we were able to implement each rule in isolation, without having to design special case logic or having to interleave the implementation of many different rules in the same contiguous textual area of code. Moreover, because the transformation function was implemented as a visitor, it was easily integrated into our existing Amalia tool suite.

Curried visitors allow many of the benefits of transformational programming without incurring the overhead of a full transformation system that comes with more general rule-based approaches to analysis, such as in the Software Refinery [18]. This paper introduces the curried visitor framework and explores the generation of curried visitors from a rule-based specification language which provides multi-parameter pattern-matching facilities. We illustrate these ideas by explaining in detail the implementation of the normal-form construction of temporal logic formulas.

2. BACKGROUND

We illustrate the motivation and use of the curried visitor framework by way of an example, which we encountered while porting Amalia to generate analyzers for linear temporal logic (ltl). Amalia is a generator framework for synthesizing light-weight analyzers of formal specification languages. The analyzers are considered light-weight because they package analysis capability into a visitor object, which allows many different analyses to be seamlessly integrated into a tool that constructs and manipulates ASTs (e.g., a CASE tool).[1] Amalia tools generate visitor objects from the rule-based semantic definition of a notation (Section 2.1). Another key characteristic of Amalia analyzers is that they are designed for *extension* and *contraction*—in the sense suggested by Parnas [15]—so that they can be easily assembled into "useful subsets" tailored to a particular analysis need. One of these useful subsets (Section 2.2) contains components that require the use of curried visitors. To understand the example requires some background in the working of Amalia, which we now explain.

2.1 Amalia tool suite

Amalia-generated analyzers compute basic behavioral information by visiting the in-memory representation of a program. Our analyses are specified in a rule-based language called *structural operational semantics* (SOS) [16], which is commonly used to reason

[1]This definition of "light-weight" differs from that in [14], where a tool is called light-weight if operates on small and easy to write specifications.

Level	Realms and contents
3	`LTS = { inferLTS[LWA], minLTS[LTS] }`
2	`LWA = { lotosLwa[LOTOS], ltlLwa[LTL] }`
1	`LTL = { ltlTerm, ltlDNF }` `LOTOS = { lotosTerm }`

Figure 1: Current elaboration of domain model.

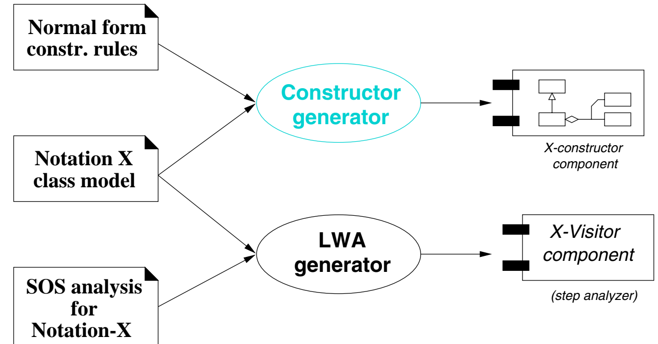

Figure 2: Amalia tool suite. Greying indicates `Constructor generator` is under development as of this writing.

about formal notations (e.g., LOTOS, CCS, and ESTEREL). Despite our success in analyzing formal specifications, we found that to implement more sophisticated program analyses requires the management of a significant amount of *analysis context* by the visitor. Unfortunately, visitors that must manage a large amount of context can be very difficult to design and to understand [19].

Amalia combines principles from the GenVoca model of component based application generators [2] with our results in visitor-based implementation of formal analyses [8] to produce efficient and flexible analyzers for formal notations. In [20], we show how analysis algorithms and methods are made light-weight by *refining* language-independent algorithms and methods to use a visitor implementation. We designed a GenVoca-style domain model[2] to capture and specify this refinement (Figure 1).

Our domain model comprises three different levels, which indicate the logical level of a component in the context of a layered hierarchy. A particular analyzer in this domain is a layered refinement—specified by a GenVoca type equation—that references components from each level in Figure 1. For example, the type equation:

```
sd = inferLTS[ltlLWA[ltlTerm]]
```

declares a component `sd`, which implements a state diagram by analyzing the AST of an ltl formula. Specifically: `ltlTerm` is a Level-1 component that encapsulates the construction of ltl ASTs; `ltlLWA` is a Level-2 component that encapsulates basic light-weight analysis of an AST using a visitor object; and `inferLTS` is a Level-3 component that encapsulates the logic for deriving the states and transitions of a state-diagram by using the light-weight analysis services provided by the Level-2 component. Any verification algorithm that can be parameterized by `LWA` can be automatically refined into a lightweight analyzer.

[2]We provide a brief summary of the GenVoca notational conventions in the appendix.

Components in the LWA realm are not written by programmers, but rather are generated automatically from the structural operational semantics of a notation. Figure 2 depicts the architecture of the Amalia tool suite. In this diagram: documents (crimped-page icons) represent input files that a human designer must construct, ellipses represent Amalia tools, and plug-ins (rectangles with protruding pins) represent generated components. The key problem in designing the `LWA` generator is to synthesize a visitor from a semantics specification that is structured as a set of axioms and inductive inference rules. We solved this problem for SOS in [8]. The present paper is concerned with a more difficult problem: how to generate visitors that implement a set of *transformation rules*.

2.2 Level-1 services

Level-1 components implement services that constitute a basic useful subset of functionality in this domain. This collection includes services for constructing and accessing ASTs for a given notation, and for testing the ASTs for *semantic* equivalence. Two ASTs may have a different syntactic structure and yet be semantically equivalent. High-level analyzers, such as state-diagram constructors, assume the ability to test two ASTs for semantic equivalence. There are many different interpretations of equivalence. Milner [13, 9.4] provides a nice discussion of this topic for process algebras; similar arguments hold for formal logics. The Amalia generator specializes general analysis algorithms by "plugging-in" components that implement different equivalence interpretations.

To be plug compatible, every component must implement a uniform interface, called the *realm interface*. Level-1 actually comprises many realms, one for each formal notation that we want to analyze. Figure 1 highlights two such realms: `LTL`, which defines Level-1 services for formulas in linear temporal logic; and `LOTOS`, which defines Level-1 services for specifications in the formal description technique LOTOS [3]. We call a specific Level-1 realm—such as `LTL` or `LOTOS`—a *constructor realm*. Each component in such a realm (e.g., `ltlTerm` and `ltlDNF` in `LTL`) implements the realm interface, which, as noted previously, contains services for constructing, accessing, and testing the semantic equivalence of ASTs.

The constructor-realm interface encapsulates the actual representation of AST nodes *and* the logic for computing the semantic equivalence of two ASTs. This encapsulation provides a powerful facility for making engineering trade-offs. For example, in the `LTL` library, `ltlTerm` maintains the property that syntactically identical specifications share the same storage in memory. In order to detect a larger number of logical equivalences, the component `ltlDNF` stores all wffs in a disjunctive-normal form. This normal form is expensive to compute in general, but is relatively cheap to maintain by construction in the absence of general negation. The two type equations:

```
sd1 = inferLTS[LWA[ltlTerm]]
sd2 = inferLTS[LWA[ltlDNF]]
```

declare two components, `sd1` and `sd2`, both of which implement a state diagram by analyzing the AST of an ltl formula. They differ, however, in the equivalence relation used to unify states in the state diagram. Consequently, `sd1` will contain more states and transitions than will `sd2`. If, for example, we wanted to apply exhaustive model-checking techniques [5, 4], then the reduction in the size of the state space could significantly improve the efficiency of the analysis. Moreover, there are some formulas for which `sd1` will produce an infinite amount of states. It was this problem that motivated us to use a normal form.

2.3 Level-1 plug compatibility

We implement Level-1 plug compatibility using techniques from object-oriented design. This approach is discussed in detail in [2, Sect. 4]. The key idea is that a realm can be defined by a *class model*[3] and implemented by *object-oriented frameworks* [10]. A class model defines a set of classes, their relationships, and their associated attributes and operations. A realm is defined by a class model that names all of the classes, objects, and associations that are available to components that are parameterized by the realm. An object-oriented framework decomposes a problem into a set of *abstract classes*, which define only abstract operations, and a collection of sets of *concrete classes*, each of which specializes the abstract classes by providing methods for the abstract operations. A realm interface can be implemented by a set of abstract classes. A given component in a realm is then just a set of concrete classes that specialize the abstract classes in the realm interface. We used this principle to implement Level-1 realms, such as `LTL`, and their components, such as `ltlTerm` and `ltlDNF`.

Figure 3 depicts a class model for the (Level-1) constructor realm `LTL`. Classes in this model are organized by *generalization* (or "kind-of") and *aggregation* (or "part-of") associations. For example, class `Wff` generalizes classes `True`, `False`, `Proposition`, `Unary`, and `Binary`; whereas class `Binary` aggregates two `Wff`s, called `operands`. Briefly, the class model defines two `Binary` operators: `And` for conjoining two operands and `Or` for disjoining them. It also defines three `Unary` operators: `Eventu`, `Hencef`, and `Negation`, which correspond to ltl temporal operators, *eventually* ($\Diamond$) and *henceforth* ($\Box$) and to logical negation ($\neg$), respectively. Intuitively, a temporal operator asserts a property of a sequence of states: The eventually operator asserts that the `operand` is true at some state in the sequence, and the henceforth operator asserts that the `operand` is true at every state in the sequence.

The implementation of a constructor-realm interface is generated automatically from the class model that specifies the realm. Aggregations in the class model cause operations[4] to be generated in the corresponding realm interface. These operations construct composite objects from parts and extract the parts from a composite object. For example, an aggregation in Figure 3 indicates that class `Binary` provides two *accessor operations*, `getFirstOperand()` and `getSecondOperand()`, each of which returns a `Wff`. Moreover, because `And` specializes `Binary` and is a leaf class in the constructor-realm interface, it provides an operation, `Instance()`, for constructing a composite `Wff` from two operand `Wff`s. A snippet of the corresponding realm interface is displayed in Figure 4. Accessor operations are abstract in the realm interface; so different components can provide their own implementations. Similarly, the implementations of the `Instance()` operations in a constructor-realm interface are defined by different components to reflect the different equivalence relations that the components implement.

The use of the `Instance` operation to construct AST instances provides a convenient idiom for standardizing the mechanism for equivalence checking across different components in a constructor realm. Clients cannot directly create an instance of an AST class, but must request instances of a particular class using the `Instance()` service provided by that class. The `Instance()` operations provided in different Level-1 components implement dif-

[3]In this paper, we use class models that are similar to Lieberherr's *class-dictionary graphs* [12] recast in a UML-conforming notation.

[4]Operations should not be confused with operators. Services in a class interface are called *operations*, and are implemented by methods in a component, whereas *operators* belong to the ltl notation and are defined in the ltl class model (`And`, `Or`, etc.).

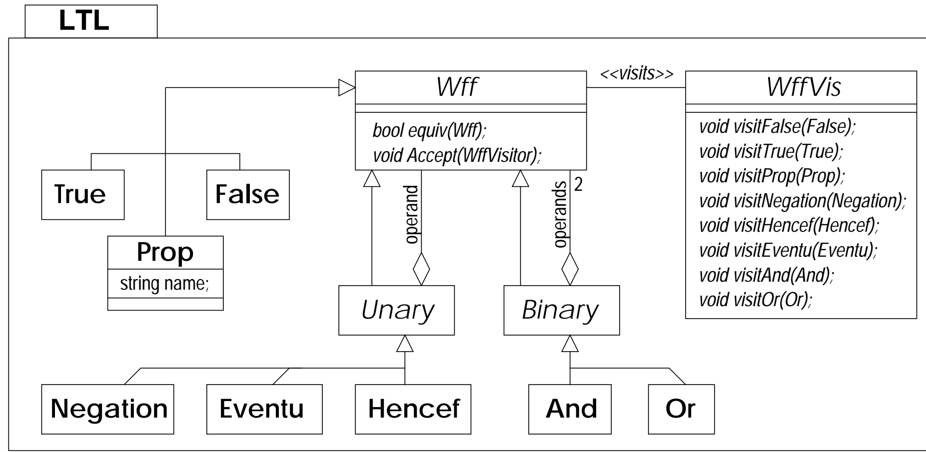

Figure 3: Framework definition of the constructor realm `LTL`.

```
class Binary :  public Wff {                    (1)
 public:                                        (2)
  virtual Wff* getFirstOperand() =0;            (3)
  virtual Wff* getSecondOperand() =0;           (4)
};                                              (5)

class And :  public Binary {                    (6)
 public:                                        (7)
  static Wff* Instance(Wff*, Wff*);             (8)
};                                              (9)
```

Figure 4: Some classes in the `LTL` constructor realm.

ferent equivalence relations and use the *singleton pattern* [9] to guarantee that only a single instance of each semantically equivalent AST is created. That is, the behavior of `Instance()` reflects the particular equivalence relation that the component implements and ensures that any two constructions of a semantically equivalent specification share the exact same storage location. This design enables higher-level components in the domain to check equivalence by comparing pointers, which greatly simplifies their implementation.

Finally, Figure 3 also shows that `Wff`s are designed to be visitable. `WffVis` is an abstract visitor class. A `WffVis` provides "call back" operations for each concrete descendant of `Wff`. In addition, `Wff` defines a polymorphic operation called `Accept`, which takes as a parameter a visitor, commensurate with the protocol of the visitor pattern [9].

3. AN EXAMPLE COMPONENT

Because some equivalences are very expensive to compute, designers often amortize the cost of checking equivalence by constructing ASTs in a canonical form. Level-1 services support such a design without violating the information hiding that enables plug compatibility. Components implement the constructor-realm interface and can be swapped in and out in order to use different canonical forms. For purposes of illustration, we describe one such component, `ltlDNF`, which serves as a running example in the remainder of the paper.

Our normal form for ltl is obtained by distributing conjunction over disjunction and generalizing conjunction and disjunction to operate on sets of `Wff`s. We refer to `Wff`s in a set of `Wff`s that are disjoined as *terms* and to `Wff`s in a set of `Wff`s that conjoined as *factors*. A `Wff` whose major operator is a temporal operator (i.e., eventually or henceforth) is called a *temporal formula*. Using this terminology, the disjunctive normal form (DNF) is defined as follows:

- An atomic formula (i.e., a `True`, `False`, or `Proposition` object, or a `Negation` object whose `operand` is a `Proposition` object) is in DNF.
- A temporal formula is in DNF if its operand is in DNF.
- A conjunction is in DNF if each of its factors is either an atomic formula or a temporal formula, and if its factors are in DNF.
- A disjunction is in DNF if each of its terms is one of an atomic formula, a temporal formula, or a conjunction; and if its factors are in DNF.

Figure 5 shows a class model that supports this DNF. Classes ending in `_impl` specialize the corresponding classes that are defined in the constructor-realm interface. For example, the class `False_impl` in the DNF class model specializes class `False` in the LTL class model, and class `And_impl` in the DNF class model specializes class `And` in the LTL class model. Shaded rectangles denote classes that are defined in the constructor-realm interface. A solid dot adorning an aggregation signifies a multiplicity of zero or more. Thus, an instance of `Or_impl` aggregates a set of `Wff`s, which represent the terms of a disjunction. An `And_impl` aggregates three sets, which identify the factors in a conjunction: `tempFacs` is a set of temporal factors, `posFacs` is the set of propositional factors, and `negFacs` is the set of propositions whose negations are factors. The partitioning of factors into these three sets simplifies local consistency checking. When building the normal form for a `Wff`, we also perform routine simplifications that can be justified by simple logical identities, e.g., $f \vee false \equiv f$, $f \wedge false \equiv false$, $f \wedge \neg f \equiv false$, and so on.

The instance graph shown in Figure 6 illustrates some sample ASTs which are in DNF. To help in reading these diagrams, we also show the formulas that they represent using a more conventional concrete syntax (in which $\Diamond$ denotes the eventually operator, $\vee$ denotes disjunction, $\wedge$ denotes conjunction, and $\neg$ denotes negation). For example, the formula $(b \wedge c \wedge \neg a) \vee a$ is a disjunction of two terms. One term (a) is a proposition; the other ($b \wedge c \wedge \neg a$) is a conjunction containing three factors: two of the factors (b and

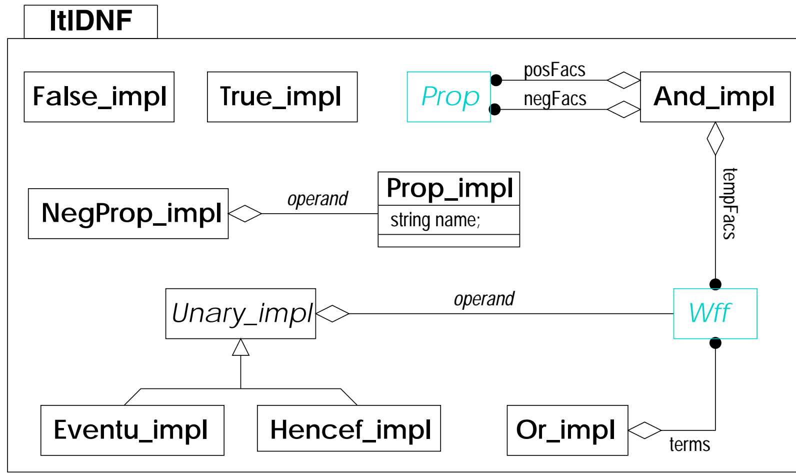

Figure 5: Class model for the `ltlDNF` component.

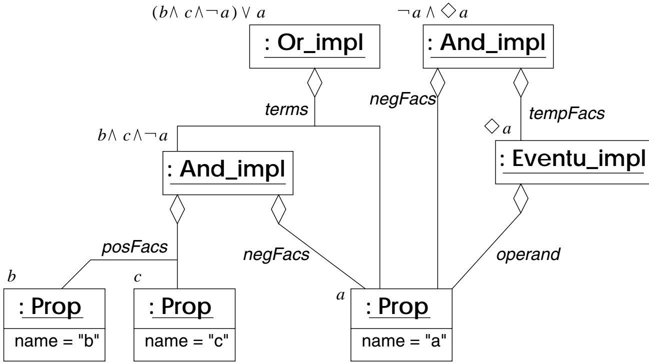

Figure 6: Instance graph showing some DNF representations.

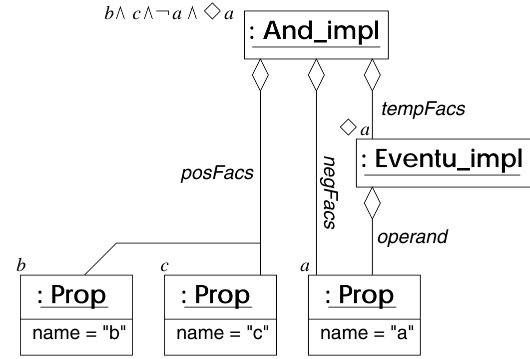

Figure 7: Result of conjoining $(b \wedge c \wedge \neg a) \vee a$ and $\neg a \wedge \Diamond a$.

c) are propositions and the third ($\neg a$) is a negated proposition. The formula $a \wedge \Diamond a$ is a conjunction of two factors: a factor ($\neg a$) whose negation is a proposition and a temporal factor ($\Diamond a$).

In general, the class model for a component must provide an implementation class for each leaf class in the constructor-realm interface; the implementation class is named by appending `_impl` to the name of the interface class that it specializes. A leaf class in a constructor component implements its own `Instance()` operation, and also implements the `Instance()` operation of its parent class in the constructor-realm interface. For example, `And_impl` implements both `And_impl::Instance()` and `And::Instance()`. The former constructs an `And_impl` object from three sets of `Wffs`, as indicated by aggregations in the class model for the component. The latter takes two `Wffs` in as arguments and constructs a `Wff` that is semantically equivalent to the conjunction of these arguments, as indicated by an aggregation (inherited from `Binary`) in the class model that defines the constructor realm (LTL). In both cases, it is assumed that the `Wffs` given as arguments are in DNF. Observe that the latter instance operation need not return an `And_impl`.

`Wffs` can be maintained in DNF by distributing conjunctions over disjunctions and gathering together the terms (factors) in consecutive disjunctions (conjunctions). Additionally, the `ltlDNF` component performs several standard simplifications. For example, consider the case of conjoining the formulas $(b \wedge c \wedge \neg a) \vee a$ and $\neg a \wedge \Diamond a$, which are depicted in Figure 6. The `ltlDNF` component builds the DNF for this conjunction as follows.

$$
\begin{aligned}
&((b \wedge c \wedge \neg a) \vee a) \wedge (\neg a \wedge \Diamond a) \\
&\quad \equiv \; ((b \wedge c \wedge \neg a) \wedge (\neg a \wedge \Diamond a)) \vee (a \wedge (\neg a \wedge \Diamond a)) && (1) \\
&\quad \equiv \; (b \wedge c \wedge \neg a \wedge \Diamond a) \vee \mathit{false} && (2) \\
&\quad \equiv \; (b \wedge c \wedge \neg a \wedge \Diamond a) && (3)
\end{aligned}
$$

Here, (1) is the result of distributing the outer conjunction over the nested disjunction; (2) is the result of collecting factors of consecutive conjunctions, which appear in both terms, and simplifying the second term, which is locally inconsistent; and (3) is the result of a simplification. Figure 7 shows the DNF representation for this conjunction.

As this example shows, maintaining normal forms by construction requires components to interleave the logic for constructing ASTs with the logic for checking the equivalence of two ASTs. Of course, such interleaving complicates the development of these components, which is why we want to generate them automatically from declarative specifications. We now describe a declarative language that allows the specification of such components.

4. TRANSFORMATION RULES

We specified the DNF construction process outlined in Section 3 using a set of transformation rules. Each rule has a signature that serves as a pattern for matching the types of ASTs to which the

```
rule DNFAnd-True-Wff ( X : True; Y : Wff ) Y

rule DNFAnd-False-Wff ( X : False; Y : Wff ) X

rule DNFAnd-And-And ( X : And; Y : And )
  if( !empty(posFacs(X) intersect negFacs(Y))
    or !empty(negFacs(X) intersect posFacs(Y)))
  then False
  else And_impl(posFacs(X) union posFacs(Y),
                negFacs(X) union negFacs(Y),
                tempFacs(X) union tempFacs(Y))

rule DNFAnd-And-Temp ( X : And; Y : Eventu )
  And_impl(posFacs(X),
           negFacs(X),
           tempFacs(X) with Y)

rule DNFAnd-Or-Wff ( X : Or; Y : Wff )
  reduce(Or,
         image(lambda t : Wff .And(t,Y),
               terms(X))
```

Figure 8: Subset of the rules used to specify the `ltlDNF` method `And::Instance(Wff*, Wff*)`.

rule is applied, and a body that specifies how to construct a new AST—the result of applying the rule. The text in the body of a rule may refer to operators and accessors in the constructor-realm interface and also to classes that are specific to a component, such as the `And_impl` operator in `ltlDNF`. Moreover, the rules language exploits a rich toolkit of functions for constructing and extracting sets and sequences, and for constructing anonymous operators (i.e., λ-abstractions) and applying operators over sets and sequences. This toolkit and its integration with AST objects was inspired by facilities in the Refine language [18], which greatly influenced our design. We now describe in detail the specification of an operation in `ltlDNF` in order to motivate the structure of a curried visitor, which allows us to directly implement such a rule-based specification.

By far, the most complex set of rules were used to specify the `ltlDNF` constructor:

```
And::Instance(Wff*, Wff*)
```

This constructor creates the normal-form conjunction of two ASTs, which are assumed to be represented in DNF. This specification comprises a total of 28 rules. Figure 8 depicts five of these rules for illustration. In each rule, the variables X and Y are the (normal form) ASTs that are being conjoined.

The rule `DNFAnd-True-Wff` matches on two ASTs `X` and `Y` when `X` is an object of class `True` and `Y` is an object of an arbitrary `Wff` class. The construction in the body is trivial: It just returns `Y`. We read this rule to say, "The result of And-ing the AST `True` with any AST `Y` is the AST `Y`." The second rule, `DNFAnd-False-Wff`, is similar.

The rule `DNFAnd-And-And` shows the use of a conditional and of a typical set-theoretic operation. The body of the rule is a conditional, which predicates a construction on the value of a boolean condition. In this case, the condition references the accessor operations `posFacs` and `negFacs` of class `And_impl`. The condition is true when the negated factors of `X` intersect the positive factors of `Y` or vice versa, indicating that a proposition is true in one AST and false in the other. If this condition is true, the conjunction of `X` and `Y` must be false; otherwise, the conjunction is an `And_impl` and its factors are the factors of `X` and the factors of `Y`. In Amalia rules, a class name in the body of a rule is interpreted as an operator in either the constructor-realm class model or the component class model. Consequently, the references to `False` and `And_impl` are interpreted as invoking the operations `False::Instance()` and `And_impl::Instance()`, respectively. The values passed into this latter `Instance()` method are formed by taking the union of the sets returned by the designated accessor operations. For example,

`posFacs(X)` **union** `posFacs(Y)`

constructs the set $\texttt{posFacs}(X) \cup \texttt{posFacs}(Y)$.

The rule `DNFAnd-And-Temp` shows another set-theoretic operation. A new `And_impl` is constructed, and the third value passed into the `Instance()` method is the set formed by inserting the AST `Y` into `tempFacs(X)`. In this rule

`posFacs(X)` **with** `Y`

constructs the set $\texttt{posFacs(X)} \cup \{\texttt{ Y }\}$. Other set operations that may be used in Amalia rules include the set-construction operations, **union**, **intersect**, and **setdiff**, and the set query operators, **subset**, **in**, and **empty**.

Finally, rule `DNFAnd-Or-Wff` illustrates support for anonymous operators and the application of an operator over a set. In DNF, an AST node is represented as a disjunction of terms, each of which is a conjunction of factors. If AST `X` is an `Or` in DNF (implying that it must be an `Or_impl`), then to create the DNF for the conjunction of `X` with another `Wff Y` requires conjoining every term of `X` with `Y`. Such manipulation is justified by the distributive law:

$$(X_1 \lor X_2 \lor \ldots X_n) \land Y \equiv$$
$$((X_1 \land Y) \lor (X_2 \land Y) \lor \ldots \lor (X_n \land Y))$$

Observe how the `Y` is distributed across each term of `X`. In Refine, distributed function application over a set (or sequence) is expressed using the **image** and **reduce** combinators, which we provide in Amalia rules. The **image** operator returns a set (sequence) of values obtained by applying a unary operator to each element of a set (sequence) and collecting the results. In this case, we create a unary operator using a lambda abstraction. The **reduce** function applies a binary operator to the elements of a set (sequence) in order to "reduce" the set (sequence) to a single composite value. In our example, the operator being applied is `Or`, the `Instance()` operation of which takes two `Wff`s as parameters. Because the result of the **image** function is a set of `Wff`s, the reduction will construct a single `Wff`.

We now turn to the automatic generation of Level-1 components from rule-based specifications, such as those presented in Figure 8. In Figure 2, the Constructor generator performs this task. This generator must address two issues. First, whereas rule patterns match on classes that are declared in the constructor-realm interface, rule bodies may reference accessors that are defined in the class model for a particular component (i.e., the `_impl` classes). This implies that an AST node, such as `X` in rule `DNFAnd-And-And` must be dynamically cast down from an `And` to an `And_impl`. The Constructor generator must consult the class models of both the constructor realm and the particular component to check the safety of these casts. Second, it should be clear that the generation of functions from these rules will be complex. One factor in this complexity is the sheer number of different cases that must be implemented. Another factor is the need to downcast the arguments

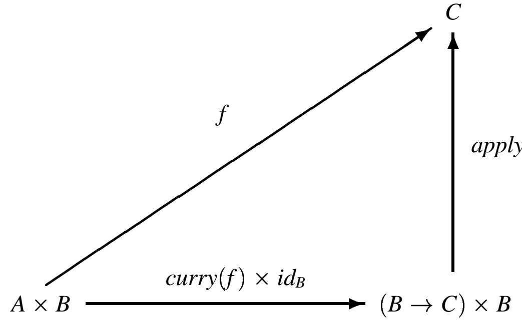

Figure 9:

to check their run-time types. Such downcasts are particularly troublesome when interleaved with the large number of cases to check. The visitor pattern obviates the need for downcasts, but only for functions over a single AST. Curried visitor frameworks extend the use of this pattern for functions over multiple ASTs. We now introduce curried visitors and demonstrate their utility as a target of code generation from Amalia transformation rules.

5. CURRIED VISITORS

A *curried visitor framework* uses multiple visitor objects to efficiently implement a function over multiple arguments of a structured data type. Because we are specifically interested in binary functions over ASTs, such as `And::Instance(Wff*, Wff*)`, we couch the discussion in these terms. Intuitively, a curried visitor framework implements a binary function with a visitor object that traverses the first argument to the function in order to *choose* an appropriate visitor with which to traverse the second argument. Upon construction, the chosen visitor records information about the first argument and then implements the binary function by traversing the second argument, referring to the stored information when necessary. A nice feature of this framework is that a particular visit method in the chosen visitor knows the run-time type of both arguments to the function. This feature greatly simplifies the manual coding of Amalia transformation rules, and we believe it will greatly simplify the automatic generation of code from such rules.

The structure of a curried visitor framework was inspired by a basic result in category theory, which relates functions over product domains to objects that can be thought of as *partially evaluated* functions formed by fixing the first coordinate of the product [5]. Figure 9 depicts this relationship for an arbitrary binary function $f : A \times B \to C$. The diagram asserts that for any such f, we can define a function $curry(f) : A \to (B \to C)$ such that $((curry(f))(a))(b) = f(a, b)$, for any $a \in A$ and $b \in B$ such that $\langle a, b \rangle \in \mathrm{dom}(f)$. The function $curry(f)$ maps an element of A to an element of the function space $B \to C$. The function *apply* maps a function $g : B \to C$ and a value $b : B$ to an element $g(b) \in C$. That is, *apply* just applies the function to an argument.

We assume that A, B, and C are structured data types, i.e., sets of objects whose classes can be defined using a class model, and that f denotes a binary function, such as `And::Instance()`. Observe that $curry(f)$ is a function with one parameter of type A. Because A is a structured date type, we can use the visitor pattern to implement

[5] Actually, category theorists call these entities *exponentials*, but we did not find this term suggestive.

```
class CurriedAndOperator :  public WffVis   (1)
{
  class AppliedToAnd :  public WffVis       (2)
   { ...  };                                (3)

  class AppliedToOr :  public WffVis        (4)
   { ...  };                                (5)

  ...                                       (6)

  CurriedAndOperator( Wff* F2, Wff*& r )    (7)
     :  f2(F2), result(r) {...}             (8)

  void visitAnd( And* f1 )                  (9)
   { AppliedToAnd vis2(f1, result);         (10)
     f2->Accept(vis2); }                    (11)

  void visitOr( Or* f1 )                    (12)
   { AppliedToOr vis2(f1, result);          (13)
     f2->Accept(vis2); }                    (14)
};                                          (15)
```

Figure 10: Elided design of the curried visitor

$curry(f)$. This visitor would need to construct a partially-evaluated function $f' \in (B \to C)$ and then *apply* f' to the argument $b : B$. That is, the visitor would have to implement

$$apply \circ (curry(f) \times id_B)$$

Given that B is also an object in a structured domain, f' can be encapsulated as a B-visitor object `bVis` and applied to an object `b` using the code:

```
b->Accept(bVis);
```

In summary, an A-visitor object, `aVis`, implements f as a side-effect of traversing an object of class A. To actually compute f, `aVis` uses a collection of B-visitor classes (henceforth referred to as `AppliedToOp`), which contains one class for each subclass `Op` of A. Intuitively, `aVis` traverses a given A-object in order to choose a particular visitor `bVis` with which to traverse a given B-object. This B-visitor then actually computes $f(A, B)$ for the given A- and B-objects.

Figure 10 illustrates (part of) the curried visitor that is used in the implementation of `And::Instance` in the `ltlDNF` component. The visitor is named `CurriedAndOperator` because it encapsulates the function *curry*(`And`), and it is a subclass of `WffVis` (Line 1). Nested class declarations (Lines 2-6) implement different partially-evaluated functions in the codomain of *curry*(`And`). These nested classes, e.g., `AppliedToAnd` and `AppliedToOr`, are also subclasses of `WffVis`, which means that each such class provides methods for operations such as `visitHencef`, `visitAnd`, and so forth. The constructor for class `CurriedAndOperator` (Lines 7-8) takes two parameters:

1. a `Wff` pointer, called `F2`, which is supplied with the second argument to `And::Instance`, and
2. a reference to a `Wff` pointer that is filled in as a side effect of the visit.

Each visit method in class `CurriedAndOperator` just creates an instance of the appropriate `AppliedToOp` class, and then applies this instance to `f2`, which stores the reference to the second argument to `And::Instance()`.

```
Wff* And::Instance( Wff* f1, Wff* f2 )  (1)
{                                        (2)
  Wff*                result=0;          (3)
  CurriedAndOperator  func(f2,result);   (4)
  f1->Accept(func);                      (5)
  return result;                         (6)
}                                        (7)
```

Figure 11: Use of curried visitors in `And::Instance`.

The use of class `CurriedAndOperator` is illustrated in Figure 11. First, we create a variable (`result`) to hold the result of the computation (Line 3). Next, we construct a `CurriedAndOperator` object (`func`) that takes as parameters to its constructor the second AST (`f2`) and a reference to the `result` variable (Line 4), which it is to fill in as a side-effect of traversing `f1`. Next, we apply `func` to `f1` by invoking `f1->Accept` with `func` as a parameter (Line 5). When this traversal returns, we expect `result` to contain a pointer to the `Wff` that is constructed.

6. DISCUSSION

Curried visitors are able to simulate currying and the application of the partially-evaluated functions for analyses specified using Amalia transformation rules, such as the set used to specify `ltlDNF`. In this example, a large part of the computation is concerned with determining the run-time types of the parameters to the *n*-ary function being curried. We know that such functions are common in program analyzers.

A benefit of a curried visitor is that it defines a unique scope that encapsulates the implementation of each transformation rule individually. For example, the method:

```
CurriedAndOperator::AppliedToOr::visitWff(
                    Wff* )
```

defines a scope in which to implement the rule `DNFAnd-Or-Wff` from Figure 8; whereas the method:

```
CurriedAndOperator::AppliedToAnd::visitAnd(
                    And* )
```

defines a scope in which to implement the rule `DNFAnd-And-And`. By defining a unique scope to house the implementation of each transformation rule, there is no need to interleave these implementations with those of the other rules. Moreover, within such a scope, we can assure (by analyzing the class models) that downcasts from, say, an `And` to an `And_impl` are safe. This allows for the easy implementation of impl-specific constructions. The establishment of these scopes simplified the manual construction of these functions, and we believe that it will greatly simplify the automatic generation of components from transformation rules.

Our example rules required matching on the root nodes of the two ASTs being combined. If there are n distinct leaf classes in the class model for an AST, then there could potentially be n^2 different cases to check. In another normal form, the situation could be worse. Consider, for example, if the choice of procedure depends not only on the types of the roots of the ASTs being conjoined, but also on the types of nodes in the AST subtrees. In the Software Refinery, such patterns are called *program schemas*, and the ability to match and handle them is one of the nice features of Refine. Curried visitors could also handle program schemas; it would require that we create more deeply nested partially-evaluated function visitors for the scopes in which such rules are appropriate.

7. RELATED WORK

We provide a new framework for building trusted and light-weight analysis components using visitor objects. Research on the use of visitors in this capacity is sparse. Seiter, Palsberg, and Leiberherr suggest similar uses to illustrate that the visitor pattern is difficult to use for analyses that require a lot of context [19]. While we agree with their assessment in general, we believe that visitor-based solutions in existing programming languages are sufficient for many analysis problems that arise in software-engineering environments. Our early results affirm their utility for analyzing specifications that use structural operational semantics [20], and the present work attests to their ability to solve the problem of simplifying expressions using normal forms.

Many of the concerns that led us to develop curried visitors have led others to develop new programming paradigms. For instance, adaptive programming facilitates design of analysis components by decoupling the logic required to traverse an AST from the processing performed during the traversal [12]. An adaptive program (AP) can concisely and transparently mirror an analysis algorithm that applies to linked data structures, such as ASTs, where these data structures are implemented independently of the AP. A *weaver* then automatically weaves the AP and the implementation of the data structures into a single program. Aspect-oriented programming is another programming paradigm that can dramatically simplify the design of analysis components. An aspect oriented program (AOP) decouples processing components from more general concerns (aspects) that cross-cut a regular program [11].

Like adaptive programming, curried visitors decouple the implementation and traversal of an AST from the processing that is performed during a traversal. Moreover, the rules language makes the specification of these analyses much more declarative than the visitor implementation, which can be generated automatically. Thus, like both adaptive programming and AOP, we advocate automated generation of efficient analysis components from high level programs. However, a weaver for an AP or an AOP generates a monolithic tangled framework, whereas our use of visitors allows us to generate frameworks that are separate from the frameworks that implement the ASTs and other analysis methods.

Finally, there has been a wealth of research into AST transformations for various purposes. TXL [6] uses transformation rules to specify how input ASTs should be transformed into output ASTs. Many of these systems weave a transformation-rule like language into the process of parsing a program. NewYacc, for example, actually augments the yacc parser generator with code that retains and annotates a parse tree so that it can later be traversed with a set of rules that fire based on the annotations [17]. Genoa extends this idea to generate stand-alone program analyzers from queries that are applied to an annotated parse tree [7]. We differ from these approaches in that, with curried visitors, we can integrate transformation capability seamlessly into existing tools, provided that they define ASTs to conform to our Level-1 realm restrictions. Because many of the decisions in Level-1 realm interfaces are consistent with good design practice, e.g., hiding the actual implementation of a data structure, we believe that these are reasonable restrictions. To date, our work on Amalia has focused on the analysis of specification languages, such as LTL and LOTOS, as opposed to real programming languages, such as C or C++. Whether or not programs in these languages are amenable to Amalia-style light-weight analysis is an open problem that we are researching.

8. ACKNOWLEDGEMENTS

We would like to thank Ramy Shahin for his helpful comments on early drafts of this document. Partial support for the first author was provided by NSF grants CCR-9901017 and EIA-0000433. Partial support for the second author was provided by NSF grants CCR-9896190 and EIA-0000433.

9. REFERENCES

[1] D. Batory. Refinements and separation of concerns. In *Proc. of the Second Workshop on Multi-dimensional Separation of Concerns*, 2000. Co-located with ICSE'2000.

[2] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Meth.*, 1(4):355–398, October 1992.

[3] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Comp. Netw. ISDN Sys.*, 14(1), 1987.

[4] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. of the* 5th *International Symposium on Logic in Computer Science*, June 1990.

[5] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite–state concurrent systems using temporal logic. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[6] J. R. Cordy, C. D. Halpern, and E. Promislow. Txl: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, January 1991.

[7] P. Devanbu. Genoa—a language and front-end independent source code analyzer generator. In *Proc. of the Fourteenth International Conference on Software Engineering*, 1992.

[8] L. K. Dillon and R. E. K. Stirewalt. Lightweight analysis of operational specifications using inference graphs. In *Proc. of the 2001 International Conference on Software Engineering (ICSE'2001)*, 2001.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley Publishing Company, Reading, Massachusetts, 1995.

[10] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, June/July 1988.

[11] G. Kiczales et al. Aspect oriented programming. In *European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

[12] K. J. Lieberherr. Object-oriented software evolution. *IEEE Trans. Softw. Eng.*, 19(4):313–343, 1993.

[13] R. Milner. *Communication and Concurrency*. Prentice Hall International Series in Computer Science. Prentice Hall, New York, 1989.

[14] G. C. Murphy and D. Notkin. Lightweight source model extraction. In *Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, October 1995.

[15] D. Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.*, 5(2), 1979.

[16] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[17] J. M. Purtilo and J. R. Callahan. Parse tree annotations. *Communications of the ACM*, 32(12):1467–1477, December 1989.

[18] Reasoning Systems Incorporated, Palo Alto, CA. *Refine User's Guide*.

[19] L. M. Seiter, J. Palsberg, and K. J. Lieberherr. Evolution of object behavior using context relations. *IEEE Trans. Softw. Eng.*, 24(1):79–92, January 1998.

[20] R. E. K. Stirewalt and L. K. Dillon. A component-based approach to building formal analysis tools. In *Proc. of the 2001 International Conference on Software Engineering (ICSE'2001)*, 2001.

APPENDIX

A. BRIEF OVERVIEW OF GENVOCA

In the GenVoca model of software generators, a data refinement is represented as a reusable, interchangeable component [2]. Each component is an instance of a type, called a *realm*. A component may be parameterized by the types of components that it uses to implement its services. Component composition, or *layering*, is then realized by instantiating one component with other components, such that each actual parameter type checks with the corresponding formal parameter. A component `b` that expects to be instantiated with components of types T_1 and T_2 is written: $b[T_1, T_2]$. Types gather plug-compatible components into libraries. For example,

```
T = { a, b[T, X], c[T] }
```

defines a library of three components, each of whose type is `T`. Component `a` does not require parameters; whereas `b` and `c` require parameters of type `T`, and `b` also requires a component of a different type (`X`). A particular instantiation of components is called a *type equation*. The type equation

```
d = c[a]
```

represents a software component `d` whose top layer is a `c`-component and that implements its services using an `a`-component (the bottom layer). Because the top layer of `d` is a `c`-component, the type of `d` is `T`, and we can add `d` to the library.

GenVoca components implement *large-scale refinements* [1], and GenVoca-component layering implements the successive refinement of an abstraction (represented by the realm of the top-layer component) in terms of another (represented by the realm of the bottom-layer component). We use this idea to automatically refine high-level verification algorithms down to work over step analyzers, which operate over in-memory representations. The GenVoca model is mature and is supported by a wealth of research into efficient implementations in modern programming languages.