Appeared in Proc. of Compiler Optimization Meets Compiler Verification Workshop.

Using Verified Data-Flow Analysis-based Optimizations in Attribute Grammars[1]

Eric Van Wyk and Lijesh Krishnan[2]

Department of Computer Science and Engineering
University of Minnesota

Abstract

Building verified compilers is difficult, especially when complex analyses such as type checking or data-flow analysis must be performed. Both the type checking and program optimization communities have developed methods for proving the correctness of these processes and developed tools for using, respectively, verified type systems and verified optimizations. However, it is difficult to use both of these analyses in a single declarative framework since these processes work on different program representations: type checking on abstract syntax trees and data-flow analysis-based optimization on control flow or program dependency graphs.

We present an attribute grammar specification language that has been extended with constructs for specifying attribute-labelled control flow graphs and both CTL and LTL-FV formulas that specify data-flow analyses. These formulas are model-checked on these graphs to perform the specified analyses. Thus, verified type rules and verified data-flow analyses (verified either by hand or with automated proof tools) can both be transcribed into a single declarative framework based on attribute grammars to build a high-confidence language implementations. Also, the attribute grammar specification language is extensible so that it is relatively straight-forward to add new constructs for different temporal logics so that alternative logics and model checkers can be used to specify data-flow analyses in this framework.

Key words: compiler optimization, optimization verification, data flow analysis, attribute grammars

1 Introduction

Building verified or high-assurance compilers is a challenging task; it is especially difficult and often beyond the abilities of current automatic proof

[1] This research is partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.

[2] Email: Van Wyk - `evw@cs.umn.edu`, Krishnan - `krishnan@cs.umn.edu`

techniques when complex program analyses must be performed to support type checking or data-flow analysis-based optimization. These analyses often require a precise understanding of the subtle ways in which program constructs interact, and thus defining them, even in declarative frameworks, is difficult. Both the type theory and program optimization communities have addressed this problem by developing techniques to prove the correctness of their respective program analyses.

To build a verified compiler one must write a specification of the compiler that addresses all aspects of the language and that is amenable to the kinds of analyses that are done to generate proofs of correctness. An obstacle to this is that different aspects of the compiler are more naturally defined using different notations for different program representations. For example, type-checking is typically a syntax-directed activity and thus the process of checking types in a program and of proving the correctness of a type system is tightly coupled to the syntax-tree program representation. On the other hand, optimizing transformations are often written as rewrite rules with data-flow analysis-based side conditions, and performing data-flow analysis and proving its correctness are tightly coupled to the control flow graph or dependency graph program representations.

In the case of type-checking, type rules used to prove the soundness (safety) of a type system follow the structure of the program's abstract syntax. Thus they can easily be transcribed into an attribute grammar [10] based implementation, thereby providing us with a high-degree of confidence that the implementation is correct. Consider the type rule for function abstraction in the simply-typed lambda calculus (taken from [17, page 103]) shown in Fig. 1(a). Γ is the *environment* that maps variable names to types. The rule states that if extending Γ with the mapping of x to type T_1 ensures that the term t_2 has type T_2, then the lambda-expression $\lambda x{:}T_1.t_2$ has the type $T_1 \to T_2$ in the environment Γ. For example, if Γ maps a variable `y` to the type `Int`, the above rule would show that the expression λ`x:Int.y+x` has type `Int->Int`. The rule (T-ABS) can be transcribed onto the `Abs` production for function abstraction shown in Fig. 1(b). Here, an inherited attribute `env` plays the role of Γ. The environment for the lambda-expression body `t2` is the environment of the lambda-expression `t` (`t.env`) extended by `envAdd` with the mapping from `x` to the type `T1`. `t2` uses its `env` attribute to compute its `type` attribute which is then used to create the functional type for the term `t` using the function `funcType`. Thus, attribute grammars (AGs) provide an

$$\frac{\Gamma, x{:}T_1 \vdash t_2{:}T_2}{\Gamma \vdash \lambda x{:}T_1.t_2{:}T_1 \to T_2} \quad \text{(T-Abs)}$$

(a)

```
production Abs
t::Expr ::= x::Id  T1::Type t2::Expr
  t.type = funcType(T1, t2.type);
  t2.env = envAdd((x,T1),t.env);
```

(b)

Fig. 1. A type rule (a) and attribute grammar implementation (b).

$$\begin{array}{c} s : v := e \Rightarrow skip \\ \text{if} \\ s \models AX \neg E[true\ U\ use(v)] \\ \text{(a)} \end{array}$$

```
production assign s::Stmt ::= v::Id e::Expr
 s.isdead = s |= AX !E[ true U v in uses];
 s.uses = e.uses ;
```
(b)

Fig. 2. Optimization rewrite rule(a) and an attribute grammar implementation(b).

attractive framework for implementing languages with complex analyses like type-checking since they provide a high-level declarative framework for specifying syntax-directed semantics of languages. Critically, implementations can be automatically generated from the AG specification.

We would also like to transcribe verified optimizations, such as the simple dead-code elimination rule in Fig. 2(a), into an attribute grammar specification of an imperative language, but this is not easily done since analyses that do not naturally follow the structure of the abstract syntax tree (AST) are difficult to specify using AGs. This rule states that the assignment statement $v := e$ on the control flow graph node s can be replaced by a skip statement if s satisfies the CTL property shown. The property is satisfied if on all successors of s, there does not exist a path to a use of the variable v. That is, v is not used in a future computation. We[3] verified the correctness of this rule in an earlier paper [13] (note that we assume expressions have no side effects). We might like to use this verified optimization in an attribute grammar to flag certain assignments as dead. To do so we would like to write a production like the one in Fig. 2(b). Here, `isdead` is a boolean-valued attribute and `uses` is an attribute containing the variables used (referenced) by the statement. Of course statements and conditions of while loops and if-then-else statements must be connected in a control flow or program dependency graph to enable a model checker to check if this formula holds on a particular assignment statement. This is not shown in the figure but in Sec. 3.1.

The primary contribution of this paper is to show how an attribute grammar specification language can be extended to allow semantic analyses based on the control flow graph of a program. Of specific interest are the data-flow analyses used in verified optimizations of imperative programs. We present extensions to an attribute grammar specification language, called Silver (see Sec. 2), that include constructs for building labelled control flow graphs and for model checking these graphs with both CTL and LTL-FV temporal logic formulas using, respectively NuSMV [4] and a hand-built LTL-FV model checker that supports free variables (Sec. 3). Thus, in one declarative formalism both syntax-tree-based and control flow graph-based analyses can be declaratively specified. This is very useful since both types of analyses are needed in building efficient high-confidence compilers. A distinguishing characteristic of this work is that Silver is designed as an *extensible language* so that writing additional extensions to Silver to use different temporal logics and model checkers

[3] Van Wyk and David Lacey, Neil D. Jones, and Carl Frederiksen.

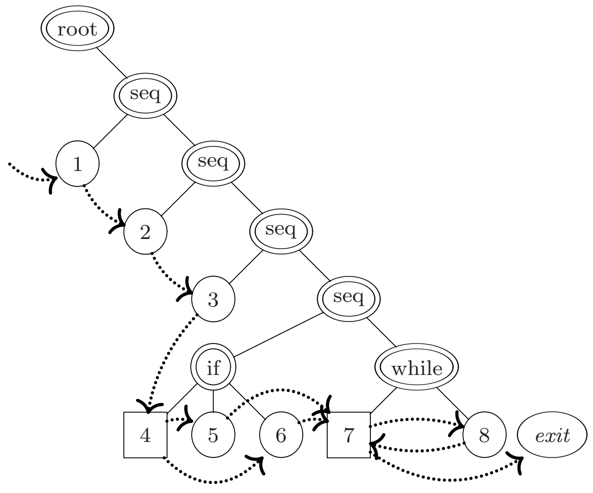
to do data-flow analysis is a straight-forward process (see Sec. 4). (Note that we have *not* built tools that read Silver specifications and prove their correctness although this is a promising direction of research). Sec. 5 provides a discussion of future and related work and concludes.

2 Attribute Grammars and Silver

In this section we present a brief introduction to our attribute grammar specification language Silver through the specification of a small subset of C (called C^-) that is used in examples in the paper. A simple C^- program is shown in Fig. 3(a). Such a small language is presented here so that a relatively complete specification that illustrates the details of Silver and the data-flow analysis extensions can be described in the available space.

```
1 {  a = 0;
2    b = 1;
3    c = 3;
4    if (a == 1)
5       b = 5;
     else
6       b = 2;
7    while (b == 3)
8       a = a + 1;
  }
```

(a)



(b)

Fig. 3. C^- program (a) and its AST and control flow graph (b)

Attribute grammars [10] specify program semantics by associating values (called attributes) with nodes in a program's abstract syntax tree (AST). Thus non-terminals are decorated with attributes and productions are associated with semantic functions that define attribute values. A partial Silver specification for C^- is shown in Fig. 4. It defines 4 non-terminals: the AST root (`Root`), statement lists (`Stmts`), statements (`Stmt`), and expressions (`Expr`), and a single terminal for identifiers (`Id`). A pretty-print attribute (`pp`) stores the constructs textual representation, has type `String`, and `occurs on` all non-terminal symbols. The `def` attribute has type `String` and stores either the name of the identifier defined by the `assign` statement or the empty string to indicate no variables are defined, as happens on `skip` and expressions. Attribute `uses` is a string list that contains the names of variables referenced by an expression or statement and thus occurs on `Expr` and `Stmt` non-terminals.

```
nonterminal Root, Stmt, Stmts, Expr;
terminal Id ;

synthesized attribute pp :: String
  occurs on {Root, Stmt, Stmts, Expr};
synthesized attribute def :: String
  occurs on {Stmt, Expr};
synthesized attribute uses :: StringList
  occurs on {Stmt, Expr};

production root
r::Root ::= s::Stmts
 r.pp = "{\n" ++ s.pp ++ "\n}\n" ;

production stmt_cons
stmts::Stmts ::= s::Stmt ss::Stmts
 stmts.pp = s.pp ++ "\n" ++ ss.pp ;

production stmt_one
stmts::Stmts ::= s::Stmt
 stmts.pp = s.pp ;

production if_then_else
i::Stmt ::= c::Expr t::Stmt e::Stmt
 i.pp = "if (" ++ c.pp ++ ...
```

```
production while
w::Stmt ::= cond::Expr body::Stmt
 w.pp = "while (" ++ cond.pp ++ ...

production assign
s::Stmt ::= v::Id e::Expr
 s.pp = v.lex ++ "=" ++ e.pp ++ ";";
 s.def = v.lex;
 s.uses = e.uses;

production skip
s::Stmt ::=
  s.def = "" ;
  s.uses = empty_str_list();

production equality
e::Expr ::= l::Expr r::Expr
  e.def = "";
  e.uses = append_str_list
             (l.uses, r.uses);

production var_ref
e::Expr ::= v::Id
  e.def = "";
  e.uses = cons_str_list (v.lex,
             empty_str_list());
```

Fig. 4. Attribute grammar specification of C^{-}in Silver

Productions define the abstract syntax of C^{-}.[4] They specify the production name and its signature which has named non-terminal and terminal symbols. Consider the `assign` production which defines assignment statements. It defines the pretty-print attribute on the `Stmt` node named `s` to be composed from the lexeme (`lex`) of the identifier terminal (the identifier's name) and the pretty-print attribute of the expression `e`. It defines `def` as the identifier's lexeme and `uses` as the `uses` of expression `e`. The other productions and attribute definitions should be self-explanatory. All of the attributes here are *synthesized* attributes and are used to propagate (synthesize) information that flows up the AST. In later sections some *inherited* attributes are defined and these pass information down the tree from parent to child nodes; these are commonly used to pass environments down the tree.

3 DFA via Model Checking in Attribute Grammars

In this section we describe extensions to Silver that allow the declarative specifications of control flow graph-based analyses, specifically data-flow analysis. These extensions allow one to write the temporal logic specifications of the

[4] For the purposes of this paper we can assume that the concrete syntax is defined using a traditional parser and scanner generator.

data-flow analysis and to invoke a model checker to perform the specified analysis. In Sec. 3.1 we show the Silver extension for building a control flow graph (CFG) and specifying what attributes label its nodes. In Sec. 3.2 we describe the CTL extension to Silver that generates a NuSMV model from the CFG and allows CTL formulas to be specified and checked using the NuSMV model checker. In Sec. 3.3 we do the same for LTL extended with free-variables and our implementation of its model checker. In Sec. 4, we describe how Silver extensions that perform CTL and LTL-FV model checking are implemented.

3.1 Building control flow graphs in Silver

The first extension to Silver allows one to easily specify how the control flow graph is built and what attributes label its nodes. This extended language is used in Fig. 5 to define the control flow graph for C^- programs. The first line of that specification indicates that some of the `Stmt` and `Expr` nodes in the AST may be used as nodes in the CFG. The CFG is specified as a distinguished set of control flow edges that overlay specified nodes in the AST. In C^-, the assignment statement nodes, the skip statement nodes, and the conditional expression nodes in the if-then-else statements and while loops will be nodes in the CFG. To build the CFG, we write additional productions, called "aspect productions" that add definitions of additional attributes to existing productions. These productions define a synthesized attribute `entry` of type `CFG_Node` that occurs on `Stmt` and `Stmts` nodes. This attribute indicates the entry node in the CFG. On the `root` production, all nodes reachable from the `stmts.entry` attribute will be the nodes in the CFG for the program.

To construct the value of the `entry` attribute we use the construct

`cfg` $\langle$*AST node for the CFG*$\rangle$ $\langle$*successor nodes*$\rangle$

The inherited attribute `succ` defines the successor in the CFG for a construct. Fig. 5 shows examples of some productions that specify CFG nodes. For example, the entry node of the CFG for the if-then-else is the conditional expression `cond` with successors that are the entry nodes for the *then* and *else* statements. This construct can be interpreted as marking the specified AST nodes as also being nodes in the CFG. The AST for the example C^- program in Fig. 3(a) is shown in Fig. 3(b), with the CFG edges indicated by dotted lines. The edge with no source indicates that node 1 is the entry node and a special exit node is also created. Conditions are boxes; statements are circles.

We also need to indicate which attributes are to be used to label the nodes of the CFG. This is necessary since we will not actually model check the CFG, but generate a representation of it to give to the NuSMV and LTL-FV model checkers. This is done using the `cfg attributes` construct. In the example in Fig 5, we have specified the `def` and `uses` attributes from Fig. 4 as CFG attributes because they are used in the data-flow properties used as examples in Secs. 3.2 and 3.3. All nodes declared as CFG nodes (`Stmt` and `Expr` in this case) must have these CFG attributes defined on them.

```
cfg nodes Stmt, Expr;

cfg attributes def, uses;

synthesized attribute entry::CFG_node
  occurs on {Stmt, Stmts} ;
inherited attribute succ::CFG_node
  occurs on {Stmt, Stmts} ;

aspect production stmt_cons
stmts::Stmts ::= s::Stmt ss::Stmts
  stmts.entry = s.entry ;
  s.succ = ss.entry ;
  ss.succ = stmts.succ ;
```

```
aspect production assign
s::Stmt ::= v::Id e::Expr
  s.entry = cfg s [s.succ];


aspect production if_then_else
s::Stmt ::= c::Expr t::Stmt e::Stmt
  s.entry = cfg cond [t.entry,
                      e.entry];
  t.succ = s.succ ;
  e.succ = s.succ ;

aspect production skip
s::Stmt ::=
  s.entry = cfg s [s.succ];
```

Fig. 5. Calls to construct the control flow graph in the specification of C^-

3.2 Checking CTL properties on the CFG using NuSMV

In this section we describe constructs that extend Silver that allow CTL formulas to be checked on the CFG and the construction of a NuSMV model from the CFG on which the formulas are actually checked.

Finite state model checkers like NuSMV require that a model specification give the finite domain of all state variables. In our example, the domains of the NuSMV state variables corresponding to the CFG attributes `def` and `uses` are the set of all the variables in the program, and, respectively, the power set of this set. These sets are constructed using the synthesized attribute `all_vars` (not shown); on the root node its value is the set of all variables in the program. Because this information is needed, the NuSMV representation of the CFG is constructed on the root of the AST. This attribute and its definition on `root` are shown in Fig. 6. The construct `smvmodel` ⟨*CFG entry node*⟩ is used to construct the NuSMV model. To build the model it gathers all the successors of the given CFG node and uses the given ranges of attributes that decorate the CFG nodes. A range is required for each CFG node attribute specified by the `cfg attributes` construct, in this case, `def` and `uses`. (A self-loop on the *exit* node is also added as required by the semantics of CTL that all states have at least one successor.) The inherited attribute `smv_model` (some definitions not shown) defined on `root` is passed down to all the AST nodes so that it can be used to do data-flow analysis.

As an example of data-flow analysis-based optimizations, we show how (immediate) dead code analysis on the `assign` production is used to determine if the assignment should be changed to a skip statement. The synthesized higher-order [23] attributes `opt_stmt` and `opt_stmts` are used to construct the optimized program. In the root production, the value of the `opt_stmts` attribute (on `s::Stmts`) is the tree representing the optimized program.

To determine if `opt_stmt` is set to skip or the original assignment statement, the `assign` production calls the model checker with a property specified

```
inherited attribute smv_model :: SMV_Model occurs on {Stmts, Stmt} ;

aspect production root  r::Root ::= stmts::Stmt
  stmts.smv_model = smvmodel stmts.entry
                    [def ranges over r.all_vars,
                     uses ranges over powerset of r.all_vars];

synthesized attribute opt_stmt  :: Stmt  occurs on Stmt ;
synthesized attribute opt_stmts :: Stmts occurs on Stmts ;

aspect production assign  s::Stmt ::= v::Id e::Expr
  s.opt_stmt = if s.smv_model, s.entry |=
    AX (   A [!(v.lex in uses) U (def == v.lex && !(v.lex in uses)) ]
        || AG !(v.lex in uses)  )
               then skip ()     else s;

aspect production stmt_cons  stmts::Stmts ::= s::Stmt ss::Stmts
  stmts.opt_stmts = stmt_cons (s.opt_stmt, ss.opt_stmts) ;
```

Fig. 6. DFA-based optimizations using CTL and NuSMV.

in CTL. The syntax of the model checking expression is $M, s \models f$ where M is the model, s is the state, and f is the formula. This expression has boolean type. In the use of this construct on the `assign` production in Fig. 6 the model M is the inherited attribute `smv_model`. The state s is the CFG node for the assignment `s`. The property f is the CTL formula which states that on all next (`AX`) states from the `assign` CFG node, either (*i*) on all paths (`A`), `v`'s lexeme `lex` is not in `uses` until (`U`) it is equal to `def` (that is, v it is not used until it is defined, with the new definition not using the old value) or (*ii*) globally on all paths (`AG`), `v`'s lexeme does not ever appear in `uses` (that is, the assigned value is not used later). This formula would be satisfied on nodes numbered 2 and 3 in Fig. 3(b). Note that this formula is different from the one presented in Sec. 1; this one finds more dead assignment statements but can also be verified.

To understand precisely how the CTL formula is model-checked we note that for each assignment statement in a program, the model checker will be called with the same model but with a different CTL formula. The formula is constructed from attribute values, such as `v.lex`. For the statement `b = 1;` in Fig. 3 the constructed CTL formula is

```
AX (   A [!(b in uses) U (def == b && !(b in uses)) ]
    || AG !(b in uses) )
```

This formula is translated to its NuSMV representation before it is passed to NuSMV to be checked. Because the attributes `def` and `uses` are not bound to an AST node, they are not instantiated and their values on the CFG nodes are used during model checking.

3.3 Checking LTL-FV properties on the CFG

In this section, we present a second extension to Silver that performs DFA via model checking. In this case, the temporal logic is LTL-FV [11], a version of LTL extended with free variables. The free variables are similar to the free variables added to CTL by Lacey and de Moor [12] and used in our proofs of optimization correctness [13,14]. Instead of returning the set of states that satisfy a model, the model checkers for these logics return a set of nodes and substitutions for the free variables that satisfy the formula. For the example LTL-FV formula in this section, this set of substitutions specifies all (immediately) dead assignment statements. Thus the model checker is called once for the model, instead of once for each assignment as in Sec. 3.2. Thus, the creation of the model and the invocation of the model checker are both done on the `root` production for C^-.

```
aspect production root  r::Root ::= stmts::Stmts
 r.ltlfv_model = ltlfvmodel stmts.entry ;
 stmts.deadResults = modelcheck r.ltlfv_model
    with (def == x) && X (  (!x in uses U (def == x && !x in uses))
                          || G !(x in uses))
    with [x ranging over r.all_vars];
```

In the production above the model for the LTL-FV model checker is created from `stmts.entry` in a manner similar to the CTL case. The attribute `deadResults` is assigned the set of substitutions that result from checking the model with the dead-code formula (essentially the LTL-FV version of the CTL version from above). In the case of the sample C^- program in Fig. 3 this set is $\{(2, [x \mapsto b]), (3, [x \mapsto c])\}$ The LTL-FV model checker is finite state, like NuSMV, and thus to generate a finite state model we specify the set of possible values for the free variables in the formula. Here, `r.all_vars` is, as before, the set of all variables.

The results of the model checker (`stmts.deadResults` in this case) are passed down as an inherited attribute to the productions that need the results. The extension provides a construct to check whether a particular substitution satisfies the formula on a particular node. The syntax of this construct is

`check` ⟨*results*⟩ `for` ⟨*substitutions*⟩ `on` ⟨*CFG node*⟩

Thus, to know if an assignment is dead, we check to see if the formula is satisfied on the `assign` production's node, with `x` substituted with `v.lex`.

```
aspect production assign  s::Stmt ::= v::Id e::Expr
 s.opt_stmt = if check s.deadResults for [x mapsto v.lex] on s.entry
              then skip ()   else s ;
```

On the AST node 2 for the program in Fig. 3 the substitution $(2, [x \mapsto b])$ would be found and the check would return true and the optimized statement is then created as before. We can model check multiple formulas like this on the `root` production, and pass the results down to appropriate nodes via multiple inherited result attributes.

4 The implementation of Silver DFA extensions

A distinguishing characteristic of this work is that Silver is designed and built as an *extensible language*. Thus it can easily be extended with new language constructs and semantic analyses of those constructs. Above we presented extensions that allow Silver to perform data-flow analysis based on two different temporal logics. Extensions to use other logics and model checkers can also be added in a straight-forward way.

Silver attribute grammar specifications are implemented by translating them to Haskell code. This Silver-to-Haskell translator is implemented as an attribute grammar written in Silver. (A good test for Silver was to write a compiler for it in itself.) The declarative, modular, and extensible nature of attribute grammars is the key to implementing the DFA extensions to Silver [22]. To add new language constructs to Silver, one adds new productions that define their concrete and abstract syntax and attribute definitions for the attributes that label the nonterminals in the abstract grammar for Silver.

The productions that define the abstract syntax of the CFG and DFA constructs also specify how these constructs translate into attribute grammar specifications written in pure, non-extended, Silver. For example, the model checking constructs (`|=` and `modelcheck`) both translate into Silver "system" calls that call the appropriate model checker. To build the appropriate model representations, each extension also writes aspect productions for the CFG building productions in the CFG extension; these have attributes that specify the translation of the CFG to the NuSMV or LTL-FV model language.

The attribute grammar specification for Silver and the attribute grammar fragments defining the CFG, CTL, and LTL-FV extensions are all combined to create a specification for an extended Silver that processes the C^{-} specification given in sections 2 and 3. In fact, both CTL and LTL-FV model checking can be used in a single programming language specification if two types of logics for DFA are desired. The translation of the DFA extension constructs to pure Silver utilizes an enhancement to attribute grammars called "forwarding" [22] that allows attribute grammars to be written in a highly modular and extensible manner.

5 Related Work, Future Work and Discussion

5.1 Related Work

5.1.1 DFA via Model Checking, and comparison to traditional approaches

Steffen's pioneering work [20] first proposed the idea of using temporal logic to specify data-flow properties and using a model checker to perform the data-flow analysis specified by a particular temporal logic formula. Several have followed his lead. Schmidt [18], for example, showed that DFA is just model checking abstract interpretations of programs. One problem with this approach is that commonly used temporal logics are propositional, making the properties specified in them program-specific. Lacey and de Moor [12] extended the logic CTL so that data-flow properties did not need to be expressed in terms of variables from a specific program but could be expressed using free variables that represented program variables or other constructs such as constants. We followed this approach in adding free variables to LTL to create LTL-FV [11] and used it in Section 3.3. This approach to DFA has the advantage that solutions to DFA problems are specified at the rather high level of abstraction of temporal logic. This is especially helpful in proving the correctness of DFA-based program optimizations (see section 5.1.2).

One critical question is which of the data-flow analysis problems solved by traditional techniques (such as iterative bit-vector algorithms) can be solved by model checking. Steffen showed that a subset of the μ-calculus temporal logic was sufficient to specify all standard bit-vector iterative data-flow analyses such as dead-code analysis [20]. Although the μ-calculus is expressive and mathematically elegant it is not easy to read, write, or analyze. Steffen developed a specification language (similar to CTL) that was less expressive but much easier to use. Formulas specified in this higher-level specification language were then translated to the lower level μ-calculus.

While the logics we have considered (CTL and LTL-FV) are representative of the logics used by most commonly-used model checkers, they are not as expressive as the μ-calculus. For example, our extensions can specify only immediate and not full (or repeated) dead code elimination. But as mentioned in Sec. 4, Silver is an extensible language, so other extensions can be developed based on different logics and model checkers such as those mentioned here. Nothing prevents us from writing another extension to Silver to use a model checker for the μ-calculus to perform any standard bit-vector DFA. Also, it should be stated that the addition of free variables to logics like CTL and LTL allows us to perform analyses such as constant propagation, for which bit vector algorithms are too restrictive and which require more general monotone DFA frameworks [9]. To perform constant propagation new attributes specifying constant assignments must be defined in addition to the `def` and `uses` attributes used above to label the CFG. Many classic data flow problems specified using temporal logics with free variables are given in the original sources for these logics [13,11].

5.1.2 Optimization verification

There has been a significant amount of work done on proving the correctness of optimizing program transformations. We devised a technique for proving the correctness of such transformations when they are specified as rewrite rules with side conditions written in CTL-FV [13,7,14]. This formed the basis of the work by Lerner, et al. [15]. They restricted the side-condition specification language to a small but commonly used subset so that the proofs of correctness could be automated. Others have also developed techniques for proving the correctness of DFA-based program transformations, *e.g.* [19].

Promising approaches based on translation validation that verify that the semantics of the optimized program match those of the original un-optimized program, *e.g.* [24], have also been explored. These approaches do not verify the translator or optimizer but instead verify its results.

5.1.3 Attribute Grammars and Data Flow Analysis

We are not the first to investigate ways to perform data-flow analysis in an attribute grammar framework. Previous work in this area falls roughly into three categories. The first category includes systems that have specific data-flow analyses hand-implemented as part of the AG system. New analysis can be added only by coding the analysis in the implementation language of the AG. One such system by Tan and Lemone [21] can compute the live-variables and available expressions data flow analyses. It is implemented in Pascal and to add new analysis one must extend the system by writing Pascal code to perform the desired DFA. The second category includes systems that use what Farrow [6] calls "ad-hoc" circular attribute definitions [2]. While these are not allowed in traditional AG systems [10], some well-defined circular attribute definitions can be evaluated by computing fixpoints over the set of attribute values. Here data-flow analyses are implemented as circular attributes: the fixpoint computation over attribute values corresponds to the fixpoints computed in model checking temporal logic formulas. Farrow formalized the evaluation mechanisms for circular attributes and the conditions under which such computations are guaranteed to terminate. Systems based on such techniques form the third category. Examples of such systems include MUG2 [8] and the more recent CRAGs [16] and APS [3] systems. These systems do not enforce Farrow's termination conditions, but rely on the person writing the AG specification to follow them. Thus one may accidentally write non-terminating data-flow analyses in these systems.

We propose a fourth category, uniquely (to our knowledge) inhabited by our system, that uses temporal logic to specify data-flow properties and model-checking to implement analyses specified by formulas. The benefits include a guarantee of termination of the DFA (since model checkers for temporal logics always terminate) and, when using a logic like CTL or LTL-FV, being able to specify DFA at a higher level of abstraction than that of circular attributes. Also, Silver can be easily extended to use logics like the μ-calculus (see Sec. 5.3) that are more expressive than CTL and LTL-FV (see Sec. 5.1.1).

5.2 Future Work

In this paper we have focused more on the specification of the data-flow analysis than on the specification of the transformations to be performed. We are currently investigating mechanisms for specifying rewrite rules like those in Fig. 1(a) so that they, along with the DFA side conditions, can be more directly transcribed into the AG specification. The approach presented in this paper uses the higher-order attribute `opt_stmt` to construct the optimized program in a bottom-up fashion. This is similar to the bottom-up application of rewrite-rules used in OPTRAN [8] to perform program transformation.

Interprocedural data-flow analysis is not supported by the two logics presented here, but we are planning to build further extensions to Silver that perform this type of analysis. There are logics and model-checkers that can be use to perform interprocedural DFA – one is CaReT [1], a logic that can encode procedure calls and returns. This Silver extension will also need to add constructs to Silver for building nodes and edges in the control flow graph that indicate calls to functions and returns from those functions. These are needed to ensure that only valid call-return paths are used in model-checking a property. Because Silver is extensible adding such analyses is straight-forward.

Other future work addresses scalability issues so that the ideas presented here can be employed on complete language descriptions. Because the control flow graph abstraction typically tracks just the "program counter" and a small set of attributes instead of the state of all program variables, the model (CFG) for programs is of reasonable size. Thus, model checkers designed to handle very large state spaces can easily handle such models. Of greater concern is the fact that in the CTL extension, NuSMV is called once for each assignment statement. Even though NuSMV is highly tuned this may be too inefficient. However, in the LTL-FV extension the model checker is called just once per analysis. But the size of the returned substitution set may be large and expensive to compute. Further investigation is required to determine if making multiple calls to fast model checkers like NuSMV is more or less efficient than making a single call to more complex model checkers for free-variable logics.

We are also investigating how to schedule and efficiently perform optimizations. Although the optimized program is easily generated using higher-order attributes (`opt_stmt` and `opt_stmts` above), the optimized tree may need to be optimized again, as in the case of dead-code analysis. This re-evaluates many attributes whose values are the same on the original and optimized ASTs. There are several avenues to pursue more efficient implementations – one is to perform an optimization in place by changing, for example, the assignment to a skip, and updating attributes incrementally [5] and only when the change affects the results of other analyses as suggested in [8].

5.3 Discussion, motivation of extensible languages

We built these data flow analysis extensions to Silver because in other research endeavors we need to write high-level language specifications that include both syntax-directed and control flow graph-based analyses in a single framework. We use Silver to write specifications for *extensible host languages* and *language extensions* that add new language constructs, semantic analyses of these constructs, and optimizations of these constructs (often based on data-flow analysis) to the host language. Many of the language extensions we have specified contain domain specific constructs. Examples include an extension that embeds SQL into an extensible Java host language and a computational geometry extension that adds efficient unbounded numeric types. We expect that language extensions will be written by people who are experts in these specific domains but are not programming language implementation experts. Thus, we want to provide a high-level formalism for specifying data-flow properties that may be used in optimizations of domain specific constructs. Temporal logics provide such a high-level formalism.

Silver is unique in that it allows, in a single declarative framework, the specification of complex semantic analyses on the program representation that is most appropriate for each analysis. Syntax directed analyses like type checking are specified in a natural way using traditional attributes and control flow graph-based analyses like data-flow analysis are specified in a natural way using temporal logic formulas. Evaluation of the analysis it carried out by either attribute evaluation or model checking. Because Silver is extensible, adding facilities for new analyses based on other temporal logics, such as the μ-calculus or CaReT as discussed above, can be done in a straight-forward manner. Thus, as new logics and model checkers are developed that are useful in performing program analyses they can easily incorporated into Silver.

Acknowledgements: We would like to thank Derek Bodin for his efforts in implementing Silver and the anonymous reviewers for their helpful comments.

References

[1] Alur, R., K. Etessami and P. Madhusudan, *A temporal logic of nested calls and returns*, in: *Proc. 10th Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, LNCS **2988** (2004), pp. 467–481.

[2] Babich, W. A. and M. Jazayeri, *The method of attributes for data flow analysis: Part I. Exhaustive analysis*, Acta Informatica **10** (1978), pp. 245–264.

[3] Boyland, J., "Descriptional Composition of Compiler Components," Ph.D. thesis, University of California, Berkeley, Berkeley, CA (1996).

[4] Cimatti, A., E. M. Clarke, F. Giunchiglia and M. Roveri, *NuSMV: A new symbolic model verifier*, in: *Proc. Computer Aided Verification*, 1999, pp. 495–499.

[5] Demers, A., T. Reps and T. Teitelbaum, *Incremental evaluation for attribute grammars with application to syntax-directed editors*, in: *ACM POPL Symp.* (1981), pp. 105–116.

[6] Farrow, R., *Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars*, ACM SIGPLAN Notices **21** (1986).

[7] Frederiksen, C. C., *Correctness of classical compiler optimizations using CTL*, in: *Proc. of Intl Workshop on Compiler Optimization Meets Compiler Verification*, ENTCS **65(2)**, 2002, pp. 1–15.

[8] Ganzinger, H., R. Giegerich, U. Möncke and R. Wilhelm, *A truly generative semantics-directed compiler generator*, SIGPLAN Not. **17** (1982), pp. 172–184.

[9] Hecht, M., "Flow analysis of computer programs," Elsevier North-Holland, 1977.

[10] Knuth, D. E., *Semantics of context-free languages*, Mathematical Systems Theory **2** (1968), pp. 127–145, corrections in **5**(1971) pp. 95–96.

[11] Krishnan, L., "LTL-FV: A Temporal Logic for Specifying Data-Flow Analyses," Master's thesis, The University of Minnesota, Department of Computer Science and Engineering, Minneapolis, MN 55455 (2005).

[12] Lacey, D. and O. de Moor, *Imperative program transformation by rewriting*, in: *10th Intl. Conf. on Compiler Construction*, LNCS **2027** (2001), pp. 52–68.

[13] Lacey, D., N. D. Jones, E. Van Wyk and C. C. Frederiksen, *Proving correctness of compiler optimizations by temporal logic*, in: *Proc. 29th ACM POPL Symp.* (2002), pp. 283–294.

[14] Lacey, D., N. D. Jones, E. Van Wyk and C. C. Frederiksen, *Compiler optimization correctness by temporal logic*, Higher Order and Symbolic Computation **17** (2003), pp. 173–206.

[15] Lerner, S., T. Millstein and C. Chambers, *Automatically proving the correctness of compiler optimizations*, in: *ACM PLDI Conf.*, 2003, pp. 220–231.

[16] Magnusson, E. and G. Hedin, *Circular reference attribute grammars - their evaluation and applications*, in: *Proc: LDTA 2003*, ENTCS **82** (2003), pp. 532–554.

[17] Pierce, B. C., "Types and Programming Languages," MIT Press, 2002.

[18] Schmidt, D. A., *Data flow analysis is model checking of abstract interpretations*, in: *ACM POPL Symp.* (1998), pp. 38–48.

[19] Shashidhar, K. C., M. Bruynooghe, F. Catthoor and G. Janssens, *An automatic verification technique for loop and data reuse transformations based on geometric modeling of programs.*, Journal of Universal Computer Science **9** (2003), pp. 248–269.

[20] Steffen, B., *Data flow analysis as model checking*, in: A. M. T. Ito, editor, *Theoretical Aspects of Computer Science (TACS'91), Sendai (Japan)*, LNCS **526** (1991), pp. 346–364.

[21] Tan, Z. and K. A. Lemone, *A research environment for incremental data flow analysis*, in: *Proc. of 13th ACM Conf. on Comp. Sci.* (1985), pp. 356–362.

[22] Van Wyk, E., O. de Moor, K. Backhouse and P. Kwiatkowski, *Forwarding in attribute grammars for modular language design*, in: *Proc. 11th Intl. Conf. on Compiler Construction*, LNCS **2304** (2002), pp. 128–142.

[23] Vogt, H., S. D. Swierstra and M. F. Kuiper, *Higher-order attribute grammars*, in: *ACM PLDI Conf.*, 1990, pp. 131–145.

[24] Zuck, L., A. Pnueli, Y. Fang and B. Goldberg, *VOC: A methodology for the translation validation of optimizing compilers*, Journal of Universal Computer Science **9** (2003), pp. 223–247.