A Flexible Architecture for Building Data Flow Analyzers

Matthew B. Dwyer
Lori A. Clarke

CMPSCI Technical Report 94–82
November 1994

Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

Data flow analysis is a versatile technique that can be used to address a variety of analysis problems. Typically, data flow analyzers are handcrafted to solve a particular analysis problem. In this paper, we describe an architecture for data flow analyzers along with a library of components and component generators that can be employed to build data flow analyzers. We further extend the architecture to support the development of analyzers for qualified data flow analysis problems. We describe our experience using this architecture to construct several prototype data flow analyzers.

This work was supported by the Advanced Research Projects Agency through Rome Laboratory contract F30602-94-C-0137.

1 Introduction

Data flow analysis is a versatile technique that has been applied to a wide variety of analysis problems. Typically, data flow analyzers are handcrafted to solve a particular problem. Building analyzers requires a significant effort. Developers must consider, and choose from, a wide variety of options for encoding the data flow analysis problem and then there is the software development effort required to implement the analyzer. While evaluation of some analysis design options can be done analytically, it is often the case that the cost-effectiveness of a particular approach must be judged empirically. In these cases, the significant software development cost of building data flow analyzers is a barrier to exploring the space of analysis design options.

In this paper, we describe a flexible architecture for building data flow analyzers that reduces software development costs. The architecture is based on Kildall's *data flow framework* [15]. The architecture defines a data flow analyzer as consisting of a function space and map, a lattice of flow values, a flow graph, and a solver. The interface to each of these is defined by an *architectural template.* Abstractions that satisfy these interfaces are referred to as *components* of a data flow analyzer.

With this architecture, a developer chooses from a collection of pre-existing components or, using high-level component generators, constructs new components and combines them to produce a data flow analyzer. A library of existing components gives developers a significant base to start from. Component generators capture common functionality and allow creation of components specialized for the problem at hand. Finally, the interfaces defined by the architecture allow existing and generated components to be reused across data flow analyzers. Thus, the cost of building, or generating, a component can be amortized over uses in a number of analyzers.

Although, we do not expect these analyzers to be as efficient as hand built, finely tuned analyzers, we provide support for the construction of high-quality implementations in the form of component generators and components that can be tuned for specific problems.

An extension of the architecture allows analyzers to be built for *qualified data flow analysis problems* [12]. The components of an existing qualified analyzer can be reused in new analyzers. This encourages an iterative approach to designing data flow analyses. A design can be experimentally evaluated, and then successive designs can incorporate additional functionality to reduce analysis cost or increase the accuracy of analysis results.

We believe this architecture supports a wide variety of analysis problems. Our motivation for building this architecture is to support rapid prototyping and evaluation of data flow analyses for use in validation and verification of concurrent software [7]. We have implemented a library that includes many general components and a number of components that are specific to this application domain. Using the architecture with this library we have constructed several data flow analyzers. These analyzers illustrate the ease with which analyzers can be implemented using the architecture.

The next section describes related work. Sections 3 and 4 describe the architecture and library of components, respectively. Section 5 describes an extended architecture for building analyzers for qualified data flow problems. Section 6 describes several data flow analyzers that have been built and our implementation of the library. We conclude with an informal evaluation of this architecture

and plans for future work.

2 Related Work

Kildall [15] introduced data flow frameworks as a means of mathematically formulating data flow problems. Subsequently, a number of theoretical results and algorithms related to data flow frameworks have been developed [16]. Algorithms for solving general data flow frameworks have been presented, e.g., [1, 11]. For restricted classes of frameworks, such as *rapid* frameworks [14], very efficient solution algorithms have been developed.

Recent work has exploited the inherent generality of data flow frameworks and attempted to explore some of the issues in supporting a flexible, general approach for constructing data flow analyzers. FIAT [10] is a framework for rapid prototyping of interprocedural analyses and transformations. It provides abstract interfaces for procedure summary information and call graphs. It also provides interfaces for describing a data flow analysis as a data flow framework and provides a general iterative solver for these analyses. Sharlit [18] is a tool for generating optimizers based on data flow analysis. It generates a data flow analyzer, based on an iterative solver, from code fragments that specify the components of a data flow framework. It provides an abstraction for the flow graph to allow generated analyzers to operate on a wide variety of representations. Sharlit supports tuning the performance of analyzers by constructing a set of rules that are used to summarize the cumulative effects of flow graph paths. These rules can greatly reduce the time and space cost of analysis. Unfortunately, they can be difficult to build and are dependent on the specified function space and, therefore, are not generally reusable. In addition, the algorithm Sharlit uses to process rules and improve the efficiency of analysis is dependent on the reducibility of the flow graph. For reducible graphs, Sharlit has been used to build a very efficient optimization phase of a high-quality compilation system.

Our work is similar to both FIAT and Sharlit in that it is based on specifying the data flow analysis as a data flow framework. Like those systems, our architecture provides interface descriptions of the function space, flow graph, and lattice. Unlike those systems, our approach also provides an interface to a solver algorithm, thereby allowing a variety of solution algorithms to be incorporated in data flow analyzers. Our approach provides a library of pre-existing components that can be used to build analyzers as well as generators for common classes of analyzer components. The benefits available from our architecture are not limited to reducible graphs. This is important to us since many representations of concurrent programs [2, 5, 7, 9, 17] are irreducible. In addition, we provide support for building analyzers for qualified data flow problems.

3 An Architecture for Data Flow Analyzers

Our architecture is based on Kildall's data flow framework. We define a data flow framework as (L, G, F, M):

$L = (V, \sqcap, \sqcup, \not\sqsupset, \top, \bot)$ $\qquad F = \{f \mid V \to V\}$

$G = (E, Start, Pred \mid E \to \mathcal{P}(E), Succ \mid E \to \mathcal{P}(E))$ $\qquad M \mid E \to F$

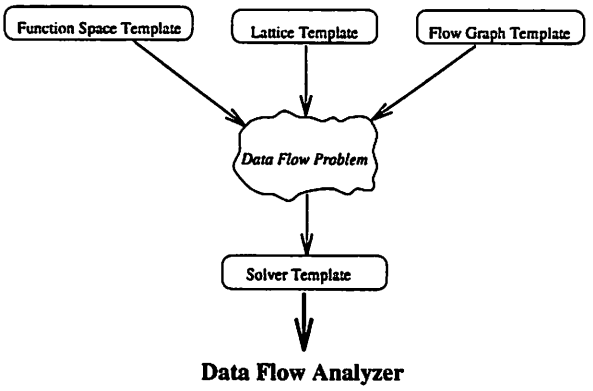

Figure 1: Architecture of Data Flow Analyzer

L is a *meet-semilattice* with a set of values, V, and the standard operators. [1] Typically, a meet-semilattice is specified only with $\sqcap$, we include $\sqcup$ and $\not\exists$ to support analyzer components that are more naturally specified using those operators. G is a *flow graph*, with a set of entities, E, a set of designated start entities, *Start*, and the predecessor, *Pred*, and successor, *Succ*, functions. This specification allows a wide variety of graph types to be viewed as a flow graph. F is a *function space* consisting of a set of *transfer functions* defined over the lattice values, V. M is a *function map* that binds flow graph entities, E, to transfer functions.

The architecture consists of templates that specify the interface to each part of a data flow framework and to the solution algorithm. In practice, function maps are defined in terms of attributes of flow graph entities; consequently, our interface merges the specification of function space and function map. We refer to a mutually consistent set of lattice, function space, and flow graph components as a *data flow problem.* A solver is instantiated with a data flow problem to produce an analyzer, as illustrated in Figure 1.

In the remainder of this section we provide a description of the interfaces specified by the architectural templates.

Lattice

The lattice values constitute the data that are propagated throughout the flow graph. These values are transformed by transfer functions and combined at merge points in the flow graph. The interface to the lattice is:

```
type LatticeValue;
function Create return LatticeValue;
procedure Destroy(v :  in out LatticeValue);
function Equal(x, y :  in LatticeValue) return Boolean;
procedure Assign(l :  out LatticeValue; r :  in LatticeValue);
```

[1] Although we require $\top$, this restriction could be lifted as long as initial values are available for the problem.

```
function NotAbove(x, y :  in LatticeValue) return Boolean;
function Meet(x, y :  in LatticeValue) return LatticeValue;
function Join(x, y :  in LatticeValue) return LatticeValue;
Top :  constant LatticeValue;
Bottom :  constant LatticeValue;
```

In this interface `LatticeValue` is V, `Meet` is $\sqcap$, `Join` is $\sqcup$, `NotAbove` is $\not\sqsupseteq$, `Top` is $\top$, and `Bottom` is $\bot$. We include constructors, destructors, assignment and equality operators to allow manipulation of lattice values in intermediate computations.

Function Space

The function space consists of a set of transfer functions that propagate and potentially transform lattice values at each flow graph entity. Although not strictly part of the function space, we include a `Confluence` operator that is applied at flow graph merge points; it is typically the lattice `Meet` operator. We include it here to support generators that produce function spaces. For example, the GenKill function space generator described in Section 4 chooses `Meet` or `Join` as `Confluence` operator depending on the kind of problem specified. The interface to a function space is:

```
function Init return LatticeValue;
function Start return LatticeValue;
function Confluence(x, y :  in LatticeValue) return LatticeValue;
function FunctionMap(e :  in Entity; v :  in LatticeValue) return LatticeValue;
```

The `FunctionMap` operator can be thought of as selecting a function from F when given an entity and applying that function to the given lattice value. We include `Init` and `Start` to specify initialization values for flow graph entities and start entities respectively; for many problems these are defined using $\top$ or $\bot$. The function space described by this template is equivalent to a set of flow equations of the form:

$$\begin{aligned} In(e) &= Confluence_{p \in Pred(e)}(Out(p)) \\ Out(e) &= FunctionMap(e, In(e)) \end{aligned}$$

where the `Confluence` operator has been extended to sets of lattice values, and In and Out are the values flowing into and out of the entity.

Flow Graph

The flow graph consists of a collection of entities and predecessor and successor functions that describe the ordering of entities. The interface to a flow graph is:

```
type FlowGraph;
type Entity;
```

```
function MaxEntity(g :  in FlowGraph) return Natural;
function GetIndex(e :  in Entity) return Natural;
function Starts(g :  in FlowGraph) return SetOfEntity;
function Predecessors(e :  in Entity) return SetOfEntity;
function Successors(e :  in Entity) return SetOfEntity;
```

This interface supports a very general view of a flow graph. For example, we can specify analyses over nodes, edges, or subsets of those, and we allow multiple start entities. In addition to the interface specified for G above, we require that each entity in a flow graph can be mapped to a unique natural number, via `GetIndex`. To minimize the amount of space used to solve the data flow problem, the "indexes" should be contiguous and lie in the range $1 \ldots$ `MaxEntity`. Indexing operators are included to support construction of analyzers that are more time and space efficient. In our experience these requirements are easily satisfied and have significant payoff.

We note that backwards analyses are specified by defining the reverse of the flow graph. This can be done by exchanging the `Predecessors` and `Successors` functions for each entity and by defining `Starts` to be the exit entities of the flow graph.

Solver

A solver template requires a data flow problem, i.e., a mutually consistent lattice, function space and flow graph. A solver component is instantiated to produce:

```
type Results;
function GetInValue(e :  in Entity; r :  in Results) return LatticeValue;
function GetOutValue(e :  in Entity; r :  in Results) return LatticeValue;
function Solve(g :  in FlowGraph) return Results;
```

The `Solve` operator computes the solution to the given data flow problem for the input graph and returns the final values for flow graph entities as `Results`. `GetInValue` and `GetOutValue` are used to retrieve results for individual entities.

4 The Library of Components

The architectural templates describe interfaces that individual components must satisfy in order to be combined to produce a data flow analyzer. Conceptually, other than the template specifications, there are no restrictions on the components that may be used to fill the roles of each template. Practical data flow analyzers usually consist of monotone function spaces, finite lattices, and flow graphs that are linear in the size of the program. In this section, we describe a library of components designed to support the production of practical analyzers. Figure 2 depicts the library organized around each architectural template. Boxes with G denote generators for components that satisfy the associated template. A solid circle denotes components that are specifically designed to work together. This is an open library; as new components or generators are developed they are incorporated into the library.

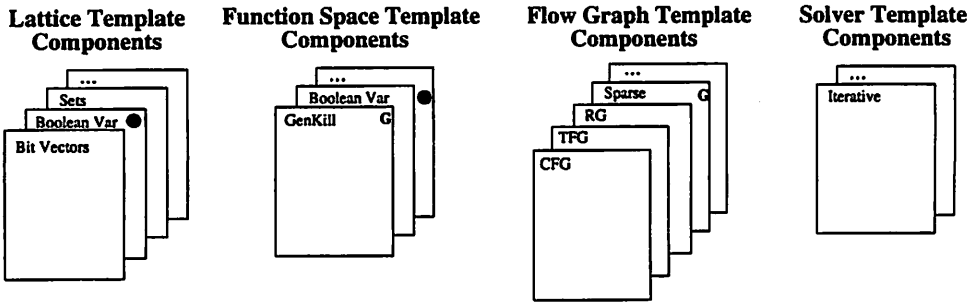

Figure 2: A Library of Components

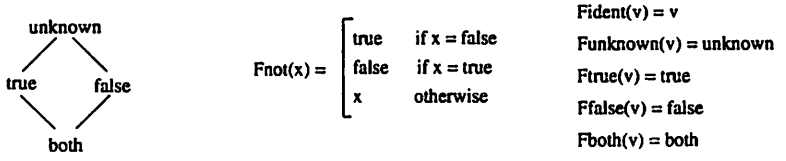

Figure 3: Boolean Variable

4.1 Lattice Components

We provide three components that satisfy the lattice interface: bit-vectors, sets and boolean variables.

Bit-vectors are a common representation in data flow problems for which the values of interest can be easily embedded in a powerset. Although they can be consumptive of space, they provide efficient `Meet`, `Join` and `NotAbove` operators. In addition, the transfer functions defined in many function spaces can be implemented efficiently as bit-vector manipulations. When the domain of values of interest is too large or the mapping from values to indexes, required for bit-vectors, is too expensive, one can use a more general set component. The boolean variable lattice component can be represented as a special case of the lattice of singletons [19]. This lattice, illustrated in Figure 3, is designed to work with the boolean variable function space to track the values of boolean variables.

We are building components for other common types of *state* variables, such as bounded counters, and for the lattice of singletons, lattice of intervals, and lattice of arithmetic congruences [8].

4.2 Function Space Components

We provide a generator for a common function space class, GenKill, and a component for tracking the value of a boolean variable.

Data flow frameworks for many classic compiler optimization problems, such as available expressions and reaching definitions, have a regular form. Individual transfer functions are constructed from a description of the values generated and killed at each flow graph entity. We provide a generator that takes as input a lattice component, an indication of whether the problem is *all paths* or *any path*, and the following:

```
function Gen(e :  in Entity) return LatticeValue;
function Kill(e :  in Entity) return LatticeValue;
```

Based on the values of the inputs, the generator produces the two major operators of a function space interface:

```
function Confluence(x,y :  in LatticeValue) return LatticeValue;
function FunctionMap(e :  in Entity; v :  in LatticeValue) return LatticeValue;
```

This provides a function space that is equivalent to flow equations of the form:

$$\begin{aligned} In(e) &= Confluence_{p \in Preds(e)}(Out(p)) \\ Out(e) &= Join(Gen(e), (In(e) - Kill(e))) \end{aligned}$$

where the `Confluence` operator is `Meet` for an *all paths* problem and `Join` for an *any path* problem.

We provide a monotone function space defined over the boolean variable lattice. The transfer functions are illustrated in Figure 3. The mapping of transfer functions to flow graph entities is made by querying attributes of entities. If the entity assigns constant `true` to the variable or the entity is the *true* branch of a conditional that tests the variable, then **Ftrue** is used. The conditions for **Ffalse** are analogous. If the entity assigns an unknown value to the variable then **Funknown** is used. If the entity assigns the negation of the variable to itself then **Fnot** is used. All other entities are bound to **Fident**, the identity transfer function.

We intend to support tracking values of other common types of *state* variables, for example bounded counters.

4.3 Flow Graph Components

A wide variety of control-flow-graph-like representations can be used directly as components in this architecture; others may require a shallow wrapper to make their interface conform. Graph types for concurrent programs such as state reachability graphs, Petri nets, and a number of control-flow-graph-like representations [2, 5, 7, 9, 17] can be adapted to meet the interface.

The cost of solving a data flow problem is strongly dependent on flow graph size. A number of representations, such as SSA [4] and the PDG [13], have been developed that effectively reduce flow graph size for some data flow analyses. Choi et. al [3] describe a general algorithm for constructing sparse data flow evaluation graphs (SDFEG) for monotone data flow frameworks. Use of this representation eliminates propagation of data through flow graph regions that add no

information to the results. Constructing an SDFEG requires finding paths through the flow graph that correspond to chains of identity, or constant, transfer functions. This dependence on the function space, however, limits reuse of an SDFEG in different data flow problems.

In contrast to SDFEGs, we provide a generator that produces a sparse flow graph component that is independent of a particular data flow problem. We use a relevance predicate to determine whether the entity is relevant and should be included in the representation, or whether it is irrelevant and should be excluded. One can think of the SDFEG construction algorithm as using a restricted relevance predicate, i.e., the existence of a non-identity and non-constant transfer function mapped to a given entity means the entity is relevant. Abstracting away from the function space allows sparse representations to be defined and reused in different data flow problems for which the relevance predicate is appropriate. The generator is given a flow graph component and a relevance predicate:

```
function IsRelevant(e :  in Entity) return Boolean;
```

It produces a sparse flow graph component and a function that is used to initialize instances of the sparse flow graph.

4.4 Solver Components

The interface specified by the solver architectural template is very general. It makes few requirements on the structure of the lattice and no requirement on the structure of the function space and flow graph. It is well know that for certain classes of data flow problems, very efficient algorithms exist. Our intent is that those algorithms are to be implemented once, installed in the library of solver components, and incorporated into data flow analyzers as needed. To date, we have provided an iterative worklist solution algorithm for frameworks with monotone function spaces [11].

5 Extending the Architecture for Qualified Analyses

The accuracy of data flow analysis suffers from the fact that all paths through the flow graph are considered executable. Encoding information about path executability can improve the accuracy of analysis results, but usually increase the size of the flow graph considerably. An alternate approach is to include information in the data flow problem that is used to restrict consideration of certain program paths. This has been done for individual data flow analyses, e.g., [19]. A more general method, and one that we employ, is to use qualified data flow analysis [12].

We refer to the data flow problem of interest as the *primary* problem. We formulate necessary conditions for path executability and encode those conditions as *constraint* data flow problems. A qualified problem is a combination of a primary and a set of constraint data flow problems. Conceptually, the qualified problem restricts the propagation of any value that violates one of the necessary conditions encoded in the constraint problems. Care must be taken at flow graph merge points so that information that could be used to restrict value propagation is not lost.

To simplify the discussion, we describe qualified analysis for a single constraint, where both the primary and constraint problems operate over the same flow graph. A qualified lattice value is a set, whose members are pairs of primary and constraint lattice values. We call such pairs *PCpairs*. The qualified function space consists of a `Confluence` operator and `FunctionMap`. We construct the qualified `Confluence` operator to preserve information that may be eventually used to restrict value flow. To enable this, the developer specifies a function, `MayDiffer`, that accepts a pair of constraint lattice values and returns true, if one of the values may cause flow to be restricted and the other value never causes flow to be restricted. The `Confluence` operator merges *PCpairs* with equivalent contraint values, as determined by `MayDiffer`. The qualified `FunctionMap` is constructed as the component-wise application of the primary and constraint `FunctionMaps`. The developer specifies a function, `TestConstraint`, that is used to restrict the set of values processed at an entity to only the input values that satisfy the constraint. The induced set of flow equations is the same as for the solver architectural template, except for the restriction operation.

$$\begin{aligned} In(e) &= Confluence_{p \in Preds(e)}(Out(p)) \\ Restricted(e) &= \{PCpair | PCpair \in In(e) \wedge TestConstraint(e, PCpair.constraint)\} \\ Out(e) &= FunctionMap(e, Restricted(e)) \end{aligned}$$

Although we could perform the restriction operation inside the `FunctionMap` and use any solver component, there are potential performance gains by using a solver component specialized for qualified data flow problems. To generate a qualified data flow problem, developers provide a flow graph, primary and constraint lattice and function space components and the following predicates:

```
function MayDiffer(x, y :  in ConstraintLatticeValue) return Boolean;

function TestConstraint(e :  Entity; v :  in ConstraintLatticeValue) return Boolean;
```

We note that analyzers for qualified data flow problems offer increased accuracy, over the primary problem, at the expense of analysis time. In general, the cost of qualified analysis is exponential in the number of constraints, so care must be taken in defining qualified problems.

The interface described above has been simplified for this presentation. We have designed a more general interface for specifying and generating qualified data flow problems. This interface allows existing qualified problems to be reused as the primary problem in specifying a new qualified analysis. This allows construction of analyzers for qualified data flow problems by incrementally composing a primary problem with a series of constraints. Using this approach, developers can try many combinations of constraints with a single primary problem to attempt to find a qualified problem whose solution meets the desired accuracy and efficiency needs.

6 Experience with the Architecture

In this section, we describe our experience building a variety of data flow analyzers and out current implementation of the library components.

6.1 Experience Building Data Flow Analyzers

To date we have built and experimented with six data flow analyzers using the architecture and associated components. We have designed a number of other analyzers, including some for qualified analysis problems. This work is part of a larger effort related to verification and validation of explicitly stated correctness properties of concurrent programs [7]. Some of the analyses are recognizable as extensions of familiar data flow analyses; others are less familiar. Our goal here is not to motivate or explain the information the algorithms produce, instead we hope to illustrate the way these analyzers were constructed from the library of existing components. We note that all of the analyzers use the iterative worklist solver component.

A trace flow graph (TFG) is a flow graph with 3 kinds of edges: control flow (CF), communication (COM) and may-immediately-precede (MIP). We built an analyzer to compute the set of communication edges in a TFG that dominate one another. The flow graph is a sparse representation of a TFG where the relevant entities are COM edges. The lattice is a bit-vector of TFG COM edges. The function space is an all paths GenKill function space. For each COM edge, `Gen` returns a vector with the edge's bit set and `Kill` returns a zero vector. Mapping of edges to functions is done using the edge index operation from the flow graph. We also built an analyzer to compute communication edge post-dominator information by reusing the same lattice and function space and the reverse flow graph.

We built a more general edge dominator analyzer. For this problem, the flow graph is a TFG where the entities are COM and CF edges. The lattice is a bit-vector of TFG COM and CF edges. The function space is a GenKill space similar to the one described above. We have also built an analyzer to compute post-dominator information by using the same lattice and function space and the reverse flow graph.

In each of these problems there were some subtleties, related to the definition of TFGs, that involved treating groups of edges as a single edge. We found that the flexibility of the component interfaces made this easy.

We built an analyzer that computes an approximate test to determine if paths in a TFG correspond to accepting strings for a deterministic finite automaton, called a PA. The flow graph is a TFG and the entities are CF, COM and MIP edges. The lattice is a bit-vector of PA states. The function space is constructed from δ, the PA state transition function. The `Confluence` operator is union and `FunctionMap`(e, v) = $\delta^*(In(e), Label(e))$, where Label(e) is a symbol in the alphabet of the PA that labels the edge. We note that δ^* is the extension of the state transition function to sets of PA states.

Our final data flow analyzer used a *TIG-based Petri Net* (TPN), a bipartite graph with node kinds called *places* and *transitions* [6]. A TPN may contain transitions that are *dead*, i.e., they are unreachable according to TPN semantics. We formulate an approximate test for *deadness* as a data flow analysis problem. The flow graph is a TPN with transitions as entities. The lattice is a bit-vector of transitions. The function space consists of distinct functions for each TPN transition. Each function tests if any predecessor has its bit set in the input value, and if so, adds the bit for the current transition. This is quite similar to the dominator function space, except that we test the input value here in a way that is not supported by the `GenKill` function space generator.

We have designed a data flow analyzer for a qualified analysis where the primary problem is state propagation and the constraint problem models boolean variable. This is useful in the analysis of concurrent programs, since it is common to have a local state variable control the pattern of inter-task communication, for example, enforcing exclusive write access. For such programs, it is often possible to improve the precision of state propagation analysis by modeling the control variable's values and restricting propagation of PA states to TFG paths that are consistent with a given value of the controlling variable. For this qualified data flow problem, `TestConstraint`(e, v) returns false if the edge, e, is a true(false) branch and the constraint value , v, is *false*(*true*), otherwise it returns true. `MayDiffer`(x, y) returns false if $x = y$ or if one value is *Both* and the other is *Unknown*, otherwise it returns true.

6.2 An Implementation

We have implemented versions of all of the components and generators described in Section 4 in Ada.

All of the generators are implemented as Ada generic packages and components are generated by generic instantiation. This has the positive benefit of making the resulting component a reusable Ada unit. Some of the more *basic* components, such as the boolean variable function space, are also implemented as generics. This allows them to be tailored to the particular application; in the case of the boolean variable function space, we instantiate it with the identity of the variable being modeled to construct the function map.

While bit operations are not directly supported in Ada, we were able to implement a bit-vector package that generates machine-level bit operations. This implementation reduces the portability of these components, however.

All of the flow graph components we used were pre-existing abstract data types implemented as Ada packages. Many of them conformed to the flow graph architectural template interface. A few required shallow wrapping to make the interfaces conform.

One drawback of Ada's type system is that there is no convenient way to encode the abstract interfaces of the architectural templates in the language. Also, Ada does not allow packages as generic parameters. This results in some syntactic inconvenience and makes the number of generic parameters large in some cases. Ada lacks first-class functions and as a result the `FunctionMap` operation essentially wraps all of the transfer functions. In practice, we found that the `FunctionMap` does not grow to be unmanageable in size.

By using Ada we were able to make use of an existing collection of reusable data types in this implementation. Ada's support for generics was crucial in being able to build a collection of components and generators that met the architectural goals of being composable and reusable. Both of these were significant factors in choosing an implementation language.

7 Conclusion

Our experience has demonstrated that it is relatively easy to construct a new data flow analyzer using this architecture. The existing library components reduce the software development costs involved in building analyzers. There are two cost reduction benefits; one does not have to write the code and one does not have to test and debug it. This may seem simplistic but in practice the payoff is high. When we needed to build new components we often found a number of opportunities for reuse. A good example is the reuse of GenKill function spaces across dominator and post-dominator problems.

We developed the four dominator related and state propagation analyzers in a matter of days. In contrast, it took approximately three weeks to develop a handcrafted analyzer for the state propagation algorithm alone. In fairness, this handcrafted analyzer was built before the others and provided valuable insight that was used in building the others. Nevertheless, the reduction in programming and testing cost gained by using the architecture and components was large.

We believe the flexibility of the architecture allows relatively efficient analyzers to be constructed. While we do not expect to obtain the performance of handcrafted analyzers, it is possible to improve the performance of selected analyzer components. For example, we intend to incorporate additional bookkeeping information in the implementation of the function space for the state propagation analyzer to eliminate redundant computation.

Our design experience with qualified analysis provides evidence that an iterative style of building data flow analyzers is beneficial, especially at the early stage of design when developers have yet to settle on the right combination of information to encode in the problem. This approach allows for rapid construction of a set of analyzers to explore the various design tradeoffs.

To support our application of data flow analysis to software validation and verification we intend to continue this work. Implementation of support for qualified analysis is underway. We plan to add new components to the library to support a number of new analyses. For example, we are interested in lattices and associated function spaces for symbolic representations of the number of times program events happen.

In summary, data flow analyses are a class of computations that have a well developed theoretical basis and a large base of interesting applications. As these applications are extended and new applications are considered, developers need support for evaluating the analysis design space. Our architecture, and associated library of components, allow low-cost construction of data flow analyzers to enable such evaluations.

Acknowledgements

A project in Kathryn McKinley's spring 1994 CS710 class inspired the GenKill function space generator.

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison-Wesley, 1985.

[2] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of event synchronization in a parallel programming tool. In *Proceedings of the Second Symposion on Principles and Practice of Parallel Programming.* ACM, 1990.

[3] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, Orlando, Florida, January 1991.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 1989.

[5] E. Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification (TAV4)*, October 1991.

[6] Matthew B. Dwyer and Lori. A. Clarke. A compact petri net representation for concurrent programs. Technical report, University of Massachusetts, 1994.

[7] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of ACM SIGSOFT'94: Symposium on Foundations of Software Engineering*, December 1994. to appear.

[8] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.

[9] Dirk Grunwald and Harini Srinivasan. Efficient computation of precedence information in parallel programs. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[10] M.W. Hall, J. Mellor-Crummey, A. Carle, and R. Rodriguez. Fiat: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[11] Matthew S. Hecht. *Flow Analysis of Computer Programs.* The Computer Science Library Programming Language Series. Elsevier North-Holland, 1977.

[12] L. Howard Holley and Barry K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, SE-7(1):60–78, January 1981.

[13] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 146–157, San Diego, California, January 1988.

[14] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23:158–171, 1976.

[15] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, Boston, Massachusetts, October 1973.

[16] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[17] S.P. Masticola and B.G. Ryder. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[18] Steven W. K. Tjiang and John L. Hennessy. Sharlit – a tool for building optimizers. *SIGPLAN Notices*, 27(7):82–93, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation.*

[19] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *Transactions on Programming Languages and Systems*, 13(2):181–210, apr 1991.