A Flexible Architecture for Building Data Flow Analyzers [1] [2]

Matthew B. Dwyer ⋆
Lori A. Clarke †

dwyer@cis.ksu.edu
clarke@cs.umass.edu

⋆Department of Computing and Information Sciences
Nichols Hall
Kansas State University
Manhattan, KS 66506
(913)532-6350

†Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 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 hand-crafted to solve a particular analysis problem. The cost of constructing analyzers can be high and is a barrier to evaluating alternative analyzer designs.

In this paper, we describe an architecture that facilitates the rapid prototyping of data flow analyzers. 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. In addition to support for traditional data flow analysis problems, this architecture supports the development of analyzers for a class of combined data flow problems that offer increased precision.

This architecture allows developers to investigate quickly and easily a wide variety of analyzer design alternatives and to understand the practical design tradeoffs better. We describe our experience using this architecture to construct a variety of different data flow analyzers.

[2] 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 hand-crafted to solve a particular problem. Building analyzers requires a significant effort; developers must consider, and choose from, a wide variety of alternative designs for encoding the data flow analysis problem and then implement the chosen design. While evaluation of some analysis design alternatives 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 comparing competing design.

Various forms of data flow analysis have been used in program optimization. Software engineering researchers have also used this technique to support software validation [CK93, DS91, FO76, OO92, TO80]. Even very different data flow analyzers share a number of common features. As has been found with domain analysis of other software areas, such as database systems, network protocols and data structures [BO92, BST+94], domain analysis of data flow analyzers can identify common reusable software components. In this paper, we describe an architecture that is built around such a collection of components. This architecture facilitates rapid prototyping of data flow analyzers by enabling specialized component generation and extensive reuse in the construction of new analyzers.

The architecture defines a set of compatible standard interfaces for components of data flow analyzers. 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. Component generators capture common functionality and facilitate creation of components specialized for the problem at hand. 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 a number of analyzers.

We envision that this architecture will be especially useful in the early stages of development of data flow analyzers. Prototype analyzers can be developed quickly for a variety of formulations of a given data flow analysis problem. After evaluating and selecting the desired problem formulation the developer may choose to create a more finely tuned analyzer. For example, a developer could decide to hand code a highly optimized version of an analyzer component or, alternatively, to selectively replace components with generated components or special purpose components that are optimized to produce a more efficient analyzer. Another strategy for improving both the efficiency and precision of analysis is combining multiple data flow analysis problems into a single problem. With combined problems, rather than executing multiple analyses independently a single analysis is run. Precision is improved when the constituent problems can use each others intermediate results. Analysis time is reduced by incurring analysis overhead costs a single time rather than once for each run of the constituent analyses. An extension of the architecture allows analyzers to be built for a useful class of combined data flow problems.

We have implemented a library of analyzer components and component generators as a collection of Ada generic packages that provide the interfaces defined by the architecture. This implementation has been used to construct a variety of data flow analyzers that are used in a toolset for analyzing explicitly stated correctness properties of distributed systems [DC94, Dwy95]. These include analyzers for *traditional* data flow problems, e.g., dominators, live variables, constant propagation, and for non-traditional problems, e.g., for complex reachability problems. These analyzers have been formulated over a variety of program representations, e.g., sequential and concurrent control flow graphs, reachability graphs for concurrent systems, Petri nets. A number of these analyzers are important components of larger software systems and have evolved over time. We have found that modifications to these analyzers have required very little programming effort. This experience validates the generality of our architecture and demonstrates the reduction in development and maintenance costs that are gained by 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 a class of combined data flow problems. Section 6 describes several data flow analyzers that have been built using an implementation of the architecture. We summarize our contributions and plans for future work in the conclusion.

2 Related Work

A typical method for describing data flow analyses is as a system of equations. These equations are derived from the structure of the program being analyzed and based on the information being computed by the analysis. An alternate means of formulating a data flow problem is as a *data flow framework*. Unlike a system of equations, a data flow framework is a collection of rigorously defined mathematical objects: a function space and map, a lattice of flow values and a flow graph. Reasoning about these mathematical objects allows one to determine performance characteristics of analyzers for a given data flow problem. A number of theoretical results and algorithms related to specific classes of data flow frameworks have been developed [ASU85, Hec77, MR90].

Most of the work on data flow analysis is aimed at supporting analysis of sequential programs. The main distinction between data flow analyses for concurrent versus sequential flow graphs is the need to have differing semantics for how values should be combined at flow graph merge points, e.g., where a node has multiple incoming edges. For example, intra-process control flow predecessors and inter-process synchronization predecessors may be treated differently. Recently data flow frameworks and solution algorithms have been extended to support analysis of concurrent programs by allowing different combining operators to be defined for different classes of nodes[Dwy95]. We incorporate this extension in our architecture.

For decades data flow analyzers have been an integral part of optimizing compilers. Well-engineered compilers often provide standard interfaces to analyzer components. These interfaces can insulate the majority of the compiler from changes to a particular data flow analyzer and may ease the integration of new analyzers in support of new optimizations. Compiler systems do not, however, provide high-level analyzer generator capabilities.

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 [HMCCR93] is a framework for rapid prototyping of interprocedural analyses and transformations; it provide interfaces for describing a data flow analysis problem as a data flow framework and provides a general iterative solver with which to construct analyzers. Sharlit [TH92] is a tool for generating compiler optimization phases that incorporate data flow analyzers; it generates a data flow analyzer, based on an iterative solver, from code fragments that specify the components of a data flow framework.

Our work is similar to both FIAT and Sharlit in that it is based on specifying the data flow analysis problem 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 treats the solver algorithm as a component of an analyzer and defines an interface for it. This allows different solution algorithms to be used in a data flow analyzer by simply changing the underlying solver. Our approach also provides a library of pre-existing components that can be used to build analyzers as well as generators for common classes of analyzer components. In addition, we provide support for building analyzers for a class of combined data flow problems.

Systems that facilitate the implementation of domain specific software systems, such as the GenVoca software system generators [BO92, BST+94], have been developed. Both GenVoca and our architecture are based on collections of parameterizable composable building blocks with standard interfaces. Consequently, we believe that programming languages designed to support GenVoca-style generators could be applied to produce an implementation of our architecture. While such support might make implementation of our architecture easier, the key ingredient in an effective domain-specific architecture is the analysis of the software domain to identify major reuseable components and candidates for component generation. Successful experience using an architecture to rapidly construct cost-effective software systems is validation of the domain analysis.

3 An Architecture for Data Flow Analyzers

Our architecture is based on the mathematical objects that constitute a data flow framework. The interface to each object is defined by an *architectural template*. Abstractions that satisfy these interfaces are referred to as *components* of a data flow analyzer. Formally, a data flow framework is (L, G, F, M, N):

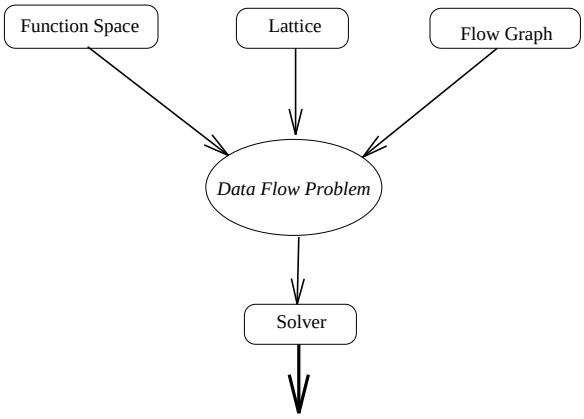

Figure 1: Architecture of Data Flow Analyzer

$L = (V, \sqcap, \sqcup, \top, \bot)$ $\quad F = \{f: V \to V\}$
$G = (E, Start, Pred: E \to \mathcal{P}(E), Succ: E \to \mathcal{P}(E))$ $\quad M: E \to F$
$N: E \to \{\sqcap, \sqcup\}$

L is a *lattice* with a partially ordered set of values, V [3]; these values encode information about program behavior that we are interested in collecting. Meet, $\sqcap$, and join operators, $\sqcup$, are included that compute for two lattice values the greatest lower bound and least upper bound, respectively. 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. Entities can be defined as subsets of flow graph nodes or edges; thereby allowing a wide variety of graphs 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; these functions encode the effects of the program on the lattice values. M is a *function map* that binds flow graph entities, E, to transfer functions. N is a *confluence map* that binds flow graph entities, E, to either the lattice meet or join operator; this operator is used to combine the values at flow graph predecessors to compute the value flowing into a given entity.

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

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

In this interface, `LatticeValue` is V, `Meet` is $\sqcap$, `Join` is $\sqcup$, `Top` is $\top$, and `Bottom` is $\bot$. We include constructor, destructor, assignment and equality operators to allow manipulation of lattice values in intermediate computations.

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;
```

In this interface, `FlowGraph` is G, `Entity` is E, `Starts` is $Start$, `Predecessors` is $Pred$ and `Successors` is $Succ$. Entities can be defined as a collection of nodes or edges; this interface allows us to view a wide variety of program representations as flow graphs. We require two additional operators, `GetIndex` and `MaxEntity`, that map entities to unique indexes and provide the maximum index value in a graph, respectively. These operators enable 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 the direction of the data flow analysis is defined by the predecessor and successor operators. For example, backward flow analyses can be defined by exchanging the `Predecessors` and `Successors` operators and by defining `Starts` to be the exit entities of the flow graph. Thus, by allowing multiple start nodes we have generalized the class of graphs to which data flow analysis can be directly applied.

Function Space

The function space consists of a set of transfer functions that propagate and potentially transform lattice values at each flow graph entity. While the operators of the function space are defined over lattice values they are not part of the lattice; there may be many function spaces defined over a given lattice. The interface to a function space is:

```
function Init return LatticeValue;
function Start return LatticeValue;
function FunctionMap(e :  in Entity; v :  in LatticeValue) return LatticeValue;
function ConfluenceMap(e :  in Entity; v1,v2 :  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. The `ConfluenceMap` operator can be thought of as selecting either $\sqcup$ or $\sqcap$ given an entity and applying that operator to the given lattice values. We include `Init` and `Start` to specify initialization values for non-start 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:

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

where, in this notation, the `ConfluenceMap` has been extended to sets, as opposed to pairs, of lattice values, and In and Out are the values flowing into and out of the entity. Note that initially for start entities $In = Start$, and for other entities $Out = Init$.

Solver

A data flow problem is formulated as a mutually consistent lattice, function space and flow graph. To produce a data flow analyzer for a problem we need to specify a solution algorithm, which we refer to as the

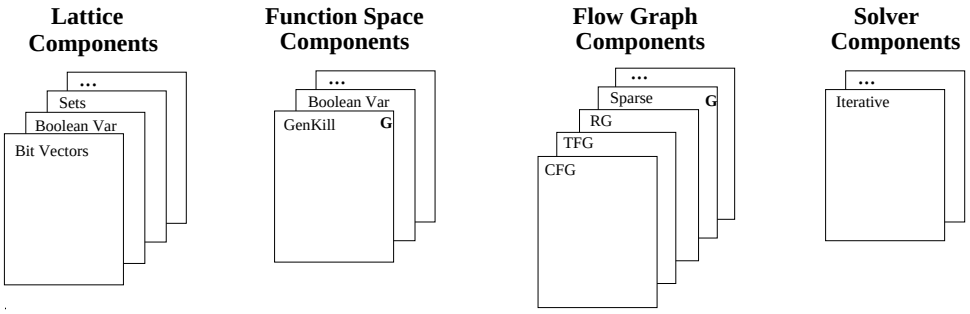

Figure 2: A Library of Components

solver component. In this architecture all solvers have a common interface that consists of a component for each part of a data flow problem. These components are given as input to the solver component and the following are produced:

```
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 a `Results` value. `GetInValue` and `GetOutValue` are used to retrieve the lattice value for individual entities. These operations constitute the interface to the data flow analyzer.

4 The Library of Components

The architectural templates define 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. We note, however, that 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. Some components are specifically designed to work together, such as the boolean variable lattice and function space.

To illustrate the construction of a data flow analyzer with the architecture and library of components we present a running example. This example is taken from a toolset for analyzing explicitly stated correctness properties of distributed systems [DC94]. In this toolset distributed systems are represented as a directed graph, called a trace flow graph (TFG). The TFG can be viewed as a collection of task control flow graphs with additional nodes and edges to represent inter-task communication. One component of this toolset transforms the TFG based on identification of sub-graphs that exhibit a particular structure, called *communication intervals*. This transformation results in a smaller graph for which subsequent analysis is more efficient and produces more precise results. A key structural feature of a communication interval is that all system executions leading into the interval execute an identifiable *entering* communication event and that all system executions leading out of the interval execute an identifiable *exiting* communication event. We present a data flow analyzer that gathers part of this information by computing pairs of communication nodes in the TFG, where one node dominates another; this *communication dominator* problem will serve as our running example. This example is implemented using a collection of Ada generic packages that constitute the library of components and generators.

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

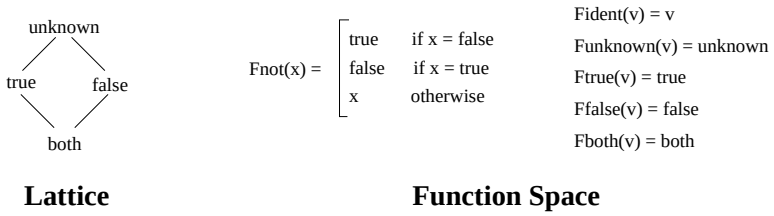

Figure 3: Boolean Variable

embedded in a powerset. Although they can be consumptive of space, they provide efficient `Meet` and `Join` 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 [WZ91]. This lattice, illustrated in Figure 3, is designed to work with the boolean variable function space to track the values of boolean variables.

The communication dominator problem uses a bit-vector encoding of the set of TFG nodes that represent communication nodes, where the `Meet` operator is bit-vector intersection and `Join` is union.

4.2 Function Space Components

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 for this common class of gen-kill function spaces. The generator takes as input a lattice component, an indication of whether the problem is *all paths* or *any path*, and the following two functions:

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

Based on the values of these inputs, the generator produces the `ConfluenceMap` operator, which for all flow graph entities is `Meet` for an *all paths* problem and `Join` for an *any path* problem [4]. The generator also produces the `FunctionMap` operator to define the function space. This provides a function space that is equivalent to the standard gen-kill flow equations:

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

We also 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.

For the communication dominator problem we define `Gen` to set only the bit corresponding to the node's index if the node is a communication node. We define `Kill` to return a bit-vector of zeros. The function space is defined as an *all-paths* problem so the `ConfluenceMap` operator will be bit-vector intersection, the `Meet` operator for our lattice. Figure 4 gives the function space definitions for this example.

[4] For clarity in the presentation we only describe the gen-kill function space generator for analyses over sequential flow graphs. A variant of the gen-kill function space generator produces a confluence map that is appropriate for analyses over concurrent flow graphs.

```
function GenNode(n :  in TFG.Node; s :  in Natural)
         return BitVector.Value is
begin
 if TFG.GetKind(n) = TFG.COM then
  return BitVector.Set(BitVector.Zero(s),
                       TFG.GetIndex(n));
 else
  return BitVector.Zero(s);
 end if;
end GenNode;

function KillNode(n :  in TFG.Node; s :  in Natural)
         return BitVector.Value is
begin
 return BitVector.Zero(s);
end KillNode;
```

```
package COMFunctionSpace is new GenKill(
   -- Bind lattice types and operators
   LatticeValue ⇒ BitVector.Value,
   Create ⇒       BitVector.Create,
   Print ⇒        BitVector.Print,
   Assign ⇒       BitVector.Assign,
   Meet ⇒         BitVector.Intersect,
   Join ⇒         BitVector.Union,

   -- Bind relevant flow graph operators
   Entity ⇒       TFG.Node,
   GetIndex ⇒     TFG.GetIndex,

   -- Indicator for all versus any path problem
   isAllPaths ⇒   TRUE,

   -- Bind Gen and Kill functions for node
   Gen ⇒          GenNode,
   Kill ⇒         KillNode);
```

Figure 4: Example Function Space

4.3 Flow Graph Components

A wide variety of control-flow-graph-like representations can be used directly as components in this architecture. Graph types for concurrent programs such as state reachability graphs, Petri nets, and a number of control-flow-graph-like representations [CKS90, DC94, GS93, MR93] can also 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 have been developed that effectively reduce flow graph size for some data flow analyses. Choi et. al [CCF91] 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, which defines graph and entity types, and start, predecessor and successor functions, and a relevance predicate:

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

It produces a sparse representation component and a constructor that builds a sparse representation from a given flow graph. The sparse representation conforms to the interface for flow graphs defined by the architecture, so that sparse representations can be used wherever a flow graph is required.

In the communication dominator problem, we could formulate the analysis over the set of all TFG nodes and use control flow predecessor and successor operators to produce a data flow analyzer. Alternatively, we could improve analyzer performance by using a sparse representation that elides all TFG nodes that are not communication nodes. Figure 5 gives the definition of the relevancy predicate, `IsCOMNode`, and sparse representation that includes only communication nodes.

```
function IsCOMNode(n :  in TFG.Node)
         return Boolean is
begin
 return TFG.GetKind(n) = TFG.COM;
end IsCOMNode;

package COMSparseTFG is new SparseRepresentation(

   -- Bind graph and entity types
   FlowGraph ⇒        TFG.Graph,
   Entity ⇒           TFG.Node,
   MaxEntity ⇒        TFG.GetMaxNodes,
   GetIndex ⇒         TFG.GetIndex,

   -- Bind graph start/end and edge operators
   IterateStart ⇒     TFG.IterateStart,
   DoneStart ⇒        TFG.DoneStart,
   GetNextStart ⇒     TFG.GetStart,
   IterateEnd ⇒       TFG.IterateExit,
   DoneEnd ⇒          TFG.DoneExit,
   GetNextEnd ⇒       TFG.GetExit,
   IteratePreds ⇒     TFG.IteratePreds,
   DonePreds ⇒        TFG.DonePreds,
   GetNextPred ⇒      TFG.GetNextPred,
   IterateSuccs ⇒     TFG.IterateSuccs,
   DoneSuccs ⇒        TFG.DoneSuccs,
   GetNextSucc ⇒      TFG.GetNextSucc,

   -- Bind the relevancy predicate
   IsRelevant ⇒       IsCOMNode);
```

```
package COMDominators is new IterativeSolver(

   -- Bind lattice type and operators
   LatticeValue ⇒    BitVector.Value,
   Create ⇒          BitVector.Create,
   Destroy ⇒         BitVector.Destroy,
   Equal ⇒           BitVector.IsEqual,
   Assign ⇒          BitVector.Assign,

   -- Bind flow graph operators
   FlowGraph ⇒       COMSparseTFG.Graph,
   Entity ⇒          COMSparseTFG.Node,
   MaxEntity ⇒       COMSparseTFG.GetMaxNodes,
   GetIndex ⇒        COMSparseTFG.GetIndex,
   IterateStart ⇒    COMSparseTFG.IterateStart,
   DoneStart ⇒       COMSparseTFG.DoneStart,
   GetNextStart ⇒    COMSparseTFG.GetStart,
   IteratePreds ⇒    COMSparseTFG.IteratePreds,
   DonePreds ⇒       COMSparseTFG.DonePreds,
   GetNextPred ⇒     COMSparseTFG.GetNextPred,
   IterateSuccs ⇒    COMSparseTFG.IterateSuccs,
   DoneSuccs ⇒       COMSparseTFG.DoneSuccs,
   GetNextSucc ⇒     COMSparseTFG.GetNextSucc,

   -- Bind function space operators
   Init ⇒            BitVector.One,
   Start ⇒           BitVector.Zero,
   Cmap ⇒            COMFunctionSpace.ConfluenceMap,
   Fmap ⇒            COMFunctionSpace.FunctionMap);
```

Figure 5: Example Sparse Representation and Analyzer Generation

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 known 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 formulated over both sequential [Hec77] and concurrent flow graphs [Dwy95].

For the communication dominator problem, the data flow analyzer is constructed using the bit-vector lattice, sparse flow graph, gen-kill function space and iterative solver. Figure 5 gives the details of this definition. We note that the solver component accepts additional parameters that are used to optionally produce detailed tracing of analyzer computation. We setup and run the analyzer by calling `COMSparseTFG.Create` with a `TFG.Graph` to create a sparse representation of the flow graph, then we pass the sparse representation as input to `COMDominators.Solve`.

5 Extending the Architecture for Combined Analyses

The precision 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 precision of analysis results, but usually increases 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., [WZ91]. A more general method, and one that we employ, is to use qualified data flow analysis [HR81].

We refer to the data flow problem of interest as the *primary* problem. We then 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. In this case, a qualified lattice value is a set, whose members are pairs [5] of primary and constraint lattice values, called *PCtuples*. We construct the qualified `ConfluenceMap` operator to preserve information that may be used to restrict value flow at some point during analysis. To enable this, the developer specifies a function, `MayDiffer`, that defines equivalence classes of constraint lattice values such that, at any point in the flow graph, either all or no members of a class cause flow to be restricted. The `ConfluenceMap` operator merges *PCtuples* with equivalent constraint values, as determined by `MayDiffer`. The qualified `FunctionMap` is constructed by applying the primary(constraint) `FunctionMap` to the primary(constraint) component of each *PCtuple* in the given qualified lattice value. The developer specifies a function, `Restrict`, 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 as follows:

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

We could move the restriction operation inside the `FunctionMap` and use any solver component. Instead, for performance reasons, we provide a specialized solver component 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 ConstraintValue) return Boolean;

function Restrict(e :  Entity; v :  in ConstraintValue) return Boolean;
```

The interface described above has been simplified for this presentation. A more general interface produces types and operators so that a qualified problems can itself fill the role of 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.

Analyzers for qualified data flow problems offer increased accuracy, over the primary problem, at the expense of analysis time. In the worst-case, the cost of qualified analysis is exponential in the number of constraints, so care must be taken in defining qualified problems. Our preliminary experience indicates that for some problems, considerable increases in the precision of analysis results can be obtained for relatively small increases in analysis time using qualified analysis.

6 Experience with the Architecture

In addition to the example presented throughout the previous sections, we have developed a number of other data flow analyzers using this architecture. In this section we describe some of those analyzers. Some solve problems that are recognizable as extensions of familiar data flow analysis problems, while others are less familiar. The analyzers use a wide variety of different flow graph, lattice and function space components. They compose both ready-made components and generated components. The degree of flexibility and large amount of component reuse that was enabled by the architecture is strong evidence that the domain analysis, component interfaces and component definitions are appropriate for the data flow analysis software domain.

Selected Analyzers

The communication dominator analyzer, presented as a running example earlier in the paper, is a traditional bit-vector problem. We were able to reuse the function space, lattice and sparse representation to generate

[5] For k constraints we would have k-tuples rather than pairs.

an analyzer for the associated post-dominator problem by switching predecessor and successor operators and defining the flow graph exit nodes as the start nodes for the data flow problem. Using the architecture, this analyzer was constructed in less than an hour using existing components and generators. This analyzer exhibits performance that is comparable to a hand-crafted implementation in this application.

After preliminary experimentation with the communication dominator analyzers we wanted to generalize the class of communication intervals that could be detected in a TFG. In doing so we were able to reuse much of the existing `COMDominator` analyzer with minor modifications. Instead of communication nodes, we required dominators and post-dominators for all *send* and *receive* nodes that are local to individual task. We redefined the `Gen` function to set appropriate bits only for send(receive) rather than communication nodes. For the new problems we consider only predecessors and successors within a given task. Task-specific predecessor and successor iterators are defined by the TFG data type. The interface to these iterators, however, does not match the flow graph interface defined by the architecture. We were able to incorporate these task-specific iterators in our analyzer by defining wrapper operators that have interfaces that satisfy the architectural definitions. The sparse representation relevance predicate was modified to check that the node kind was send(receive) rather than communication. Finally, the generation of the gen-kill function space, sparse representation, and the instantiation of the iterative solver component for these problems was the same as for the communication dominator problems. It took less than an hour to design and carry out this modification and the resultant analyzers work as expected.

We also constructed a data flow analyzer for a non-traditional problem. This analyzer computes an approximate test to determine if paths in a TFG correspond to accepting strings for a deterministic finite automaton that represents an event sequencing specification. This is called *state propagation* analysis [Dwy95]. The problem is formulated over TFG nodes connected by the set of control flow edges and an additional set of inter-task edges. The lattice is a bit-vector encoding of sets of finite automaton states. The function space is constructed from δ, the automaton state transition function. The `ConfluenceMap` operator is union for nodes with incoming control flow edges and intersection for nodes with incoming inter-task edges. The `FunctionMap`(n, v) $= \delta^*(In(n), Label(n))$, where Label(n) is a symbol in the alphabet of the finite automaton that labels the TFG node. We note that δ^* is the extension of the state transition function to sets of finite automaton states.

We have also designed a data flow analyzer for a qualified analysis where the primary problem is state propagation and the constraint problem models a boolean variable. This is useful in the analysis of distributed systems, 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 systems, it is often possible to improve the precision of state propagation analysis by modeling the control variable's values and restricting propagation of finite automaton states to TFG paths that are consistent with a given value of the controlling variable. For this qualified data flow problem, which uses the constraint problem lattice and function space illustrated in Figure 3, `Restrict`(n, v) returns false if the node, n, is a true branch(false branch) and the constraint value , v, is *false*(*true*). Intuitively, `Restrict` returns true if a value is inconsistent with a particular program execution state as defined by a TFG node. `MayDiffer`(x, y) returns false if $x = y$ or if one value is *Both* and the other is *Unknown*, otherwise it returns true.

We have prototyped a data flow analyzer that is capable of finding *dead* transitions in a *TIG-based Petri Net* (TPN) [DCN95]. The problem is formulated over the set of TPN transitions. The lattice is a bit-vector encoding of the set of transitions. The function space consists of a distinct function 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 spaces, except that we test the input value here in a way that is not supported by the `GenKill` function space generator.

In addition, the architecture has been used in a system to compute general def-use anomalies and to perform analysis of state sequencing properties over the reachability graph of concurrent programs.

7 Conclusion

The architecture has been used to develop a wide variety of data flow analyzers. Members of our research group have used it for analysis of both sequential and concurrent programs and for a variety of different internal program representations including control flow graphs, both node-based and edge-based, finite state

automata, Petri nets, and reachability graphs. The architecture has been used to formulate problems with classical gen-kill function spaces and with non-traditional functions and combinations of functions. The ability of the architecture, with its library of components and component generators, to accommodate this wide range of problems is evidence of its utility and flexibility.

Our experience to date suggests that architectural support for constructing data flow analyzers is especially beneficial at the early stages of design when developers have yet to settle on the right combination of information to encode in the problem. We were able to rapidly prototype a variety of data flow analyzers allowing evaluation of a number of analysis design alternatives. This allowed us to tune our formulation of the data flow analyzer based on observations from running the analyzer with real programs as input.

Through reuse, the existing library of components and component generators reduce the software development cost involved in building analyzers. There are two cost reduction benefits: developers do not have to write the code and do not have to test and debug it. This may seem simplistic but in practice the payoff is high. Our experience bears this out. For example, we developed the state propagation analyzer in a matter of days. In contrast, it took approximately three weeks to develop a hand-crafted analyzer for the state propagation algorithm. In fairness, this hand-crafted analyzer was built first and provided valuable insight that was used in defining the components of the analyzer generated with the architecture. Nevertheless, the reduction in programming and testing cost gained by using the architecture and components was large.

Generality is the key to supporting component reuse across different data flow analyzers. As we built each new data flow analyzer, we found a number of opportunities for reuse. A good example of such reuse is the use of the gen-kill function space and the sparse representation in both communication dominator and post-dominator problems.

Surprisingly, we also found that the flexibility of the architecture facilitated the construction of efficient analyzers. Compatible component interfaces make replacing one component with another very easy. Thus, after we had gained first hand experience using an analyzer, we were able to identify less-efficient components and easily replace them with optimized versions. By selecting and generating components that are appropriate for a given data flow analysis problem we were often able to generate analyzers that match the performance of hand-crafted analyzers.

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. We are also 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 [Gra89].

In summary, we have identified classes of components of data flow analyzers, defined general interfaces to such components and created an architecture in which such components can be easily combined to produce an analyzer for a given data flow problem. We have validated the appropriateness of this architecture by constructing a variety of different data flow analyzers with it. This work is further evidence that high-quality software systems can be generated at low cost by exploiting knowledge about a software domain.

Acknowledgements

A project in Kathryn McKinley's Spring 1994 CS710 class inspired the GenKill function space generator. The authors would like to thank Tim Chamillard, Gleb Naumovich, Peri Tarr, Sandy Wise and Dan Rubenstein for contributing to the development of the system and for experimenting with the architecture.

References

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[BO92] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *IEEE Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[BST+94] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca model of software-system generators. *IEEE Software*, pages 89–94, September 1994.

[CCF91] 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.

[CK93] S.C. Cheung and J. Kramer. Tractable flow analysis for anomaly detection in distributed programs. In *Proceedings of the European Software Engineering Conference*, 1993.

[CKS90] 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.

[DC94] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. *Software Engineering Notes*, 19(5):62–75, December 1994. Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering.

[DCN95] Matthew B. Dwyer, Lori. A. Clarke, and Kari L. Nies. A compact petri net representation for concurrent programs. In *Proceedings of the 17th International Conference on Software Engineering*, pages 147–157, April 1995.

[DS91] E. Duesterwald and M.L. 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.

[Dwy95] Matthew B. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs*. PhD thesis, University of Massachusetts, September 1995.

[FO76] Lloyd D. Fosdick and Leon J. Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys*, 8(3):305–330, September 1976.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.

[GS93] 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.

[Hec77] Matthew S. Hecht. *Flow Analysis of Computer Programs*. The Computer Science Library Programming Language Series. Elsevier North-Holland, 1977.

[HMCCR93] 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.

[HR81] L. Howard Holley and Barry K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, SE-7(1):60–78, January 1981.

[MR90] T.J. Marlowe and B.G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[MR93] 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.

[OO92] K.M. Olender and L.J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, January 1992.

[TH92] 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.*

[TO80] R.N. Taylor and L.J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions on Software Engineering*, SE-6(3):265–277, May 1980.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *Transactions on Programming Languages and Systems*, 13(2):181–210, apr 1991.