Architectural Tradeoffs for a Meaning-Preserving Program Restructuring Tool

William G. Griswold, Member, IEEE, and David Notkin, Member, IEEE

Abstract—Maintaining the consistency of multiple program representations—such as abstract syntax trees and program dependence graphs—in a program manipulation tool is difficult. This paper describes a hybrid software architecture for a meaning-preserving program restructuring tool. Layering is the primary architectural paradigm, which successively provides increasingly integrated and unified abstract machines to implement the tool. However, layering does not provide adequate control over extensibility or the independence of components. Consequently, we also adopted the paradigm of keeping the key program abstractions separate throughout the layering, providing independent “columns” of abstract data types. A pair of columns is integrated by a mapping column that translates elements in one column’s data type into related elements in the other column’s data type. Thus integration of function and separation of representation can be achieved simultaneously in this complex domain.

This hybrid architecture was crucial in overcoming severe performance problems—classic in traditional layered systems—that became apparent once the basic tool was completed. By taking advantage of the independence of the columns and the special characteristics of meaning-preserving restructuring, it was possible to extend one representation column of the architecture to the uppermost layer to provide the required access for efficient update without compromising independence. The cost of the extended architecture is that the upper layers are no longer as simple because they expose operations that only guarantee consistency under careful usage. However, the structural constraints of the hybrid architecture and provided models for building the more complicated layers minimize the negative impact of this tradeoff.

Index Terms—Software architecture, software design, program restructuring, program representation, modularization, layered systems, evolution.

I. INTRODUCTION

Layered software systems are one of the oldest architectural specimens. For example, over 25 years ago, Dijkstra designed the THE operating system [1] as a layered architecture. Since then layered architectures have been used in other operating systems, network protocols, etc.

Layering is an attractive architecture for a number of reasons. Among its benefits, Garlan and Shaw include the ability to replace implementations that maintain the same layer specifications and the ability to “design based on increasing levels of abstraction” [2, p. 10]. Dijkstra includes bottom-up implementation and testing amongst the benefits of layering [1]. Parnas stresses the benefits with respect to extension and contraction of a system: each layer “is a useful subset of the system” [3, p. 131].

One problem frequently noted about layered architectures is the difficulty of achieving satisfactory performance. Habermann, Flon and Cooprider observed: “One of the arguments against the ‘THE’ design is the overhead associated with interlevel communication among processes” [4, p. 267]. In general, “considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations” [2, p. 10]. In the context of operating systems, Stankovic explored when and how layering can be compromised to improve performance [5].

This paper is a case study exploring the tradeoffs faced in trying to achieve good performance in a layered architecture we designed for a meaning-preserving program restructuring tool [6], [7]. This tool is characteristic of tools that need to efficiently manage the consistency of independently represented abstractions (i.e., views) of the same program. These abstractions include abstract syntax trees (AST’s), control flow graphs (CFG’s), and program dependence graphs (PDG’s) [8]. Other tools that may use multiple abstractions include optimizing and parallelizing compilers, program slicing and merging tools, and programming environments.

A software engineer responsible for developing a tool in this class faces three competing challenges. One, building such a tool is a complicated task, in large part because it is hard to ensure that the data underlying the diverse abstractions are kept consistent with each other. Two, since these tools are modified over time, it is essential to capture their designs in abstractions that isolate changes within components. Three, the realization of these abstractions must be efficient enough to ensure that the end-user is satisfied with the performance of the tool.

We initially selected a layered architecture to overcome the first two challenges. Layering was used to separate the definition of the abstractions from the maintenance of consistency among the abstractions, meeting the first challenge. Using Parnas’s guidelines for anticipating subsets and supersets of function partially met the second challenge [3]. The second challenge also led us to emphasize modularity to maintain the independence of the realizations of abstractions such as the AST and PDG. To increase the independence of the AST and PDG modules, we introduced a third module to encapsulate the management of the relationships between them.
Performance was not a central concern in the basic architecture; we were more interested in demonstrating the feasibility of automated assistance for meaning-preserving program restructuring.

However, once we had proved the feasibility of the concept, the tool’s poor performance became an issue, and we needed to meet the third challenge—preferably without sacrificing the benefits of our layered architecture. We found that we could modify the architecture in principled ways to improve performance. In particular, we chose to maintain the independence of the realizations of key data abstractions throughout the layers of the tool. The tradeoff was to increase the complexity of the software components at the highest layers of the system; this cost, however, was contained by performing extensions—rather than significant rewrites—of those components.

This paper documents three lessons about layered software architectures.

- Although layering provides important benefits, performance is easily compromised without additional architectural planning; this is not a new lesson, but it’s one that can be costly to forget.
- Vertically dividing the system into columns—modules—of independent abstractions (but interdependent function) provides a principled approach to decreasing the dependencies within a layer, thus improving the architect’s ability to extend the architecture in smaller increments. A pair of independently developed modules is integrated with a “mapping” module that knows how to maintain the consistency of the data abstractions and maps data elements between them.
- The addition of columns and mappings to a layered architecture helps to separate—and hence simplify—meeting the dominant system requirements of correctness and good performance. In particular, after an initial implementation, independent increments of function can address the need for good performance without increasing the dependence between modules. However, the higher layers of the system may become somewhat more complex to use.

The way we have managed the tradeoffs between simplicity and performance in our layered architecture seems to represent a useful balance, given the state of the art. For example, our approach also has been applied successfully to the development of a program understanding tool [11]. In this application, designing with columns, layers, and mappings has been leveraged to reduce the paging overhead caused by the processing of large programs.

II. RESTRUCTURING

Some background on the restructuring paradigm helps to motivate the problems of constructing a tool that needs to maintain consistency among multiple program abstractions. A semantics-based program manipulation tool can be useful because it assists with changing some properties of a program while guaranteeing others remain unchanged. For instance, poor software structure is manifested by the number of modules that must be examined or modified to perform a single coherent change [12]. One solution to this problem is to restructure the system to improve the locality of the changes being made, but without affecting the input–output behavior of the program. However, restructuring is a complex task because the textual changes required to make a structural change are dispersed throughout the system, yet must be kept consistent with each other to preserve the original behavior of the software. Automated assistance of program restructuring can overcome this limitation [6], [7]. A tool based on this technique takes a locally-specified structural change of the program from the tool user and performs this change in conjunction with additional compensating changes throughout the program to ensure that the functional behavior of the program does not change. If the tool cannot make meaning-preserving compensating changes, the tool user’s change is disallowed and the problem is reported to assist the user in circumventing it with other transformations. Opdyke and Johnson have defined a similar approach, called refactoring, that focuses on restructuring object-oriented programs [13]; one common transformation in their approach is to replace a use of inheritance with a use of aggregation.

In general the application of a meaning-preserving transformation performs three generic actions: 1) check to see if the transformation can preserve meaning, 2) transform if so, 3) report failure if not. The check is a set of primarily semantic queries that ensure that the planned transformations on the objects will be sufficient to preserve meaning. These access the tool’s internal abstractions of the program such as an AST or PDG, depending upon which abstraction most easily provides the information.

The tool supports the restructuring of programs written in the imperative programming language Scheme, a LISP dialect. Scheme was chosen in part for its simple design and syntax. The tool is implemented in common lisp (CL) and the common lisp object system (CLOS). The PDG implementation is a subsystem of Curare [14]. It supports interprocedural analysis, including the aliasing properties of list structure references [15]. The PDG supports all features of Scheme except eval, first-class functions, continuations, and dynamic scoping. With a more complete PDG implementation the tool could support these features. The entire system contains about 22 000 lines of code. The AST module accounts for about 6700 lines, the PDG module accounts for 13 000 lines, and the mapping module accounts for the remaining 2300 lines. The performance enhancement accounts for about 1800 lines of the system, with 1100 allocated to the PDG module, and 700 to the mapping module.

III. ARCHITECTURAL OVERVIEW

A. Concepts

The layered architecture for the first version of the restructuring tool is shown in Fig. 1. The lowest layer contains the implementations—representations and algorithms—of the basic program abstractions, and exports the necessary abstract operations to the next layer. Each higher layer adds a desired service or constraint on the relationships and use of the
Because of layering, a module does not export a single monolithic interface. Rather, each of its submodules exports an interface for clients of the module.2 Due to the restrictions imposed by layering, a submodule’s exported functions can only be called by a client submodule residing in the layer above. Representing these constraints more explicitly helps ensure that two related modules are integrated properly.

The architectural interleaving of layers and modules draws primarily from the experiences of Habermann, Cooprider, and Fion with FAMOS [4]. As discussed below, they recognized that layers and modules are distinct entities, providing virtual machines and information hiding, respectively. Our interleaving also draws from Parnas, who observed that structuring to preserve the noncircularity of the uses relation—which can be thought of as separating a system into layers—can be considered independently from the structuring of modules for information hiding [3], [17]. Clark's upcalls technique exploits a similar observation about processes versus layers [18]. Ossher's grids allow for similar separations of modules and layers [19], although grids are intended for descriptive, rather than prescriptive, purposes. Rajlich's orthogonal architecture [20] shares characteristics with our approach, but the columns do not clearly revolve around data abstraction. The use of a "mediating" mapping module to increase independence of data abstractions in such an architecture appears to be unique to our approach.

As in our architecture, the FAMOS architecture was motivated by the desire to preserve the benefits of information hiding without compromising performance. For example, the process module and the segment module in FAMOS each require each other’s services. To avoid circularity of reference or a violation of information hiding, each module is split into submodules across nonadjacent layers (see Fig. 2). Consequently, the uses relation is kept noncircular and the nonadjacent layers of the same module can share representation without violating information hiding. Although Habermann, Cooprider, and Fion do not state how submodules are composed to form a layer, their treatment of a layer as an abstract machine suggests that our approach is similar to theirs. Unlike our approach, it appears that the FAMOS architecture allows only submodules to call another submodule in the layer as long as noncircularity is preserved. The FAMOS architecture also implies that one data abstraction module (e.g., the process module) makes direct calls to another (e.g., the segment module), meaning that the mapping between their respective data elements is managed directly by those abstractions. Because of the complex relationships among our program abstractions, folding the integration module into the program abstraction modules would have increased the complexity of the modules and reduced their independence.

B. Realization

In our use of this architecture, the bottom layer of the restructuring tool consists of the implementations of the basic

---

1A programming language typically has syntactic support for modules, but not layers. Consequently, the component at the intersection of a module and a layer is syntactically distinguished as the subcomponent of a module, hence the term submodule, as opposed to sublayer.

2The public interface of a submodule is a subset of the operations that are exported to the next higher submodule within the module. This approach is analogous to the way some object-oriented languages, such as C++, [16], provide different access privileges to the subclasses of a class versus its clients.
program abstractions: an AST, CFG, and PDG. The CFG and PDG reside in a single submodule, denoted for brevity as the PDG. Additionally, mappings between data elements of the AST and PDG abstractions are supported in this layer.

To ensure that the AST and PDG abstractions represent views of the same program, the next layer implements the required consistency. Because the AST is the primary abstraction seen and manipulated by the tool user, consistency is implemented by propagating AST changes to the other modules. The consistency layer receives notification of AST changes via an event announcement mechanism.

Although the consistency layer builds mappings and enforces consistency, the task of implementing restructuring transformations is still complicated by the need to access multiple program abstractions when acquiring the information needed to correctly change the program. To overcome this complication, the unified program representation layer combines the interfaces of the individual abstractions into a unified, uniform interface. For instance, it provides both syntactic queries (such as those about scoping) and semantic queries (such as those about the flow of values from one variable to another). Operations for making local updates to the AST are also provided. These are responsible for signaling such changes through event announcements so that dependent data abstractions (i.e., the mappings and PDG) can be updated. Using events rather than calls to propagate such information decreases the dependence of the AST module on the other modules.

The global transformation layer combines the queries and local updates of the unified program representation layer to implement global restructuring transformations and the semantic checks for their meaning-preserving application. Finally, the restructuring layer combines the checks and transformations into meaning-preserving operations. This layer also registers

the consistency submodule with the AST event announcements and controls when consistency is enabled. Because the restructuring transformations are in their own layers, the unified program representation layer can support additional tools for program manipulation [21], [22] or analysis [23], [24].

Despite the advantages of the basic architecture, the layering and modularity encourage an implementation that reconstructs the PDG from scratch after each restructuring transformation. However, because the group of AST changes comprising a transformation preserve meaning in a stylized way, the aggregate change to the PDG is small. The question, then, is how should the design and implementation of the tool be evolved to take advantage of this property?

There are at least two choices for improving performance: replacing the basic PDG construction algorithm with a generic incremental update algorithm that exploits the quiescence of small changes to a representation [25], [26], or exporting extra operations from the lower layers to the restructuring layer to allow efficient, but complex, updates of the PDG [27]. The latter was chosen because it is more efficient and accommodates general aliasing (which generic incremental update does not yet support [28], [29]). On the other hand, the technique only applies to semantically-constrained (e.g., meaning-preserving) transformations, so a different solution is required for other kinds of manipulation.

Because of the independence fostered by the columns and the use of events, the "efficient" architecture (see Fig. 3) is nearly a pure extension of the basic architecture. Submodules are added to the unified program representation and global transformation layers to implement the efficient updates to the PDG. Additionally, the restructuring submodule is modified to no longer register the consistency submodule for the low-level AST update events. In its place, the restructuring submodule is extended to call the efficient PDG updates.

IV. THE BASIC ARCHITECTURE

This section describes the responsibilities of the layers in the restructuring tool, and how those responsibilities are met without compromising the independence of the columns. The layers are described bottom up, showing how the facilities of each lower layer are combined to provide a more expressive and controlled layer. The reader less interested in the details of each layer in the architecture may skip to the following section, which addresses the issue of efficient update of the PDG, explaining the tradeoffs for two possible solutions and the details of the solution chosen for implementing the restructuring tool.

A. The Representation Layer

The representation layer contains the implementation of a collection of program abstractions, each of which allowing a certain class of program relationships to be efficiently or concisely manipulated. At this layer, the abstractions are completely independent of each other; the exported operations manipulating one underlying representation have no effect on the other representations at this layer. The basic abstractions are an AST, CFG, and PDG. Since the CFG and PDG are
encapsulated by the PDG module, they will be collectively referred to as the PDG. The PDG operations provided by the representation layer to the next layer are referred to as the PDG abstraction. Likewise, the exported AST operations constitute the AST abstraction, although the AST operations provided by higher layers extend this initial abstraction.

To lay the foundation for higher layers, the representation layer also provides an abstraction for mapping between associated program elements of the AST and PDG. This mapping submodule has no knowledge about the interfaces provided by the AST and PDG modules. It merely provides the kinds of mappings (e.g., one-to-one, one-to-many) required by the mapping submodule in the next layer to properly map between AST and PDG objects.

1) The AST: The AST is a widely used program data abstraction in programming environments and compilers. It closely matches a program's textual representation, providing a convenient abstraction of the program's syntactic relationships and the semantics of scoping [30]. It is also a convenient abstraction for making source code modifications.

In the restructuring tool, the AST is created from a simple top-down parse of the Scheme program's s-expressions. Each node in the AST is a representation of a program construct, such as a variable, function definition, or expression. Composite constructs are represented as nodes with children; for example, the children of a function call are the function (or name of the function) to be called and the expressions that constitute the arguments.

2) The PDG: The PDG submodule implements the CFG and PDG data abstractions. Both are derived from a program's source or AST. The CFG and PDG abstractions are used in the restructuring tool to reason about data and control relationships between program components.

The CFG is a set of vertices that are the primitive operations of a program, and a set of directed edges that represent the flow of control between the vertices [30]. A PDG [8], [31] is a set of vertices that represent the primitive operations in the program, and a set of directed edges between vertices that represent dependencies such as the flow of data between two operations. Different kinds of dependencies are represented by different kinds of edges. For example, a control dependence edge represents the effect of a conditional predicate on an operation's execution.

The first step in their construction is to translate a program (see Section IV-B) into sequences of three-address statements. The statements are translated into the CFG by replacing jumps and labels with explicit edges linking basic-blocks of these triples. The PDG models an entire program's dependencies, so a CFG is first built for each procedure, and then they are linked together into a single full-program CFG. Finally, building the PDG involves computing all the data and control dependencies for each statement, and then storing these relationships in a PDG node along with the statement [14].

3) The AST–PDG Mappings: The mapping submodule supports systematic mapping of a data element of one abstraction into its equivalent element in another abstraction. Although the AST and PDG abstractions are entirely decoupled in this layer, when they are kept consistent by the next layer, the mappings can be used by higher layers to navigate to the abstraction (and underlying representation) best suited for a particular query.

The AST–PDG mapping submodule manages two invertible relations between AST objects and PDG objects. The primary one maps between AST nodes denoting variable (and literal) references and CFG variable (and literal) references in a CFG three-address statement (see Fig. 4).

The second relation maps between AST subtrees denoting expressions and CFG variable references. This relation is used to find the data flows associated with the value produced by an expression in the AST, which are represented in the PDG by temporary variables (or temporaries). Likewise, if a search in the PDG yields a temporary, this relation maps it back to the AST expression that creates that result. The variable and expression relations cannot be put into a single relation, because mapping an AST variable as an expression yields the CFG temporary it is assigned to, but sometimes the actual CFG reference of the AST variable is desired.

The two relations are packaged as three functions: find-pdg-vars, which maps an AST variable reference to its associated CFG variable references; find-pdg-expr-vars, which maps an AST expression to its associated CFG tempo-
rury variable definition; and find-ast-expr, which maps back from a CFG variable reference to the associated AST expression. A hash table is a suitable representation for the relations, although other methods are possible [6].

Maintaining relations separately from the AST and PDG abstractions yielded two expected benefits [9], [10]. First, to implement the AST-CFG parsing procedure and the variable relation, neither the AST nor CFG themselves had to be changed. Second, it eased the evolution of relations. Originally just the variable relation was implemented. Later the expression relation was added, and soon after it was changed from a 1-1 relation to two 1-many relations. None of the cases required changing the AST or PDG implementations.

B. The Consistency Layer

For the AST, PDG, and the mappings to be useful, the data in each must be kept consistent with each other—when the AST is changed, the PDG must be updated to reflect the changes—and the mappings must be reestablished after the AST and PDG are made consistent. These requirements are met by the consistency layer.

The primary operation provided by this layer is ast-to-pdg, which is invoked by a higher layer whenever it is important that the PDG be consistent with the AST. The operation traverses the AST and creates a sequence of three-address statements, which are later passed to the PDG submodule to complete the translation. The AST-PDG mappings are also established by this operation; whenever a CFG variable is created from an AST variable, the relation is recorded in the mapping. Section V considers alternative designs that incrementally reestablish AST-PDG consistency.

Depending on how the AST's translation to statements is implemented, searches in the PDG can be difficult to perform. In particular, once an AST expression is mapped to a PDG node, it can be complicated to distinguish a graph search initiated on a node's operation and any one of its arguments. Consequently, each AST variable (and literal) reference is translated into a CFG reference that is assigned to a temporary before being used in an operation, thus giving each AST variable reference its own three-address statement (contrary to what is shown in Fig. 4).3

Besides establishing consistency and the mappings, the layer exports AST and PDG operations. All of the AST operations of the representation layer are exported, as are all the query (read-only) operations of the PDG. The PDG update operations are hidden (in this first architecture) since all PDG updates are handled by the consistency layer.

C. The Unified Program Representation Layer

Because the AST is the central abstraction upon which syntax, scoping, and program transformation depend, it is a familiar abstraction for the transformation programmer, it is desirable to have a complete, AST-oriented interface for performing all queries and transformations. This interface—the unified program representation layer—in essence combines the best of the AST and PDG in a uniform manner. Necessary parts of the AST and AST-PDG mapping submodules are imported from the consistency layer. A secondary role of this layer is supporting queries of the form "What if...?" Such a query is not simple because it almost always asks questions about the effects of introducing a construct that does not yet exist.

Also, this layer adds a submodule supporting local changes to the AST, such as deletions, insertions, and replacements. Unlike the low-level versions of these operations, when one of the change operations is called, it announces its change using the event mechanism. The AST-PDG consistency submodule is registered with the AST (by the restructuring layer) for these events, so that consistency can be maintained.

1) Translating Queries: The semantic query operations of the unified program representation layer are designed to abstract away the complexities of using the mappings and the PDG interface, but the performance and simplicity of their implementations are expected to benefit from the rich mathematical basis of data flow analysis and PDG's. The unified version of a PDG operation $Q_{pdg}$ ideally looks like

$$Q_{ast}(s) = m^{-1}(Q_{pdg}(m(s)))$$

where $m$ maps one or more AST objects to its PDG objects (i.e., find-pdg-vars and find-pdg-expr-vars), and $m^{-1}$ maps one or more PDG objects to its AST objects (i.e., find-ast-expr).

As an example, consider the query get-uses, which retrieves all the uses of a variable assignment in the AST. Since the "uses" information is not readily available in the AST, the implementation of get-uses uses the def-use information that is stored in the uses attribute of CFG variable definitions. To get access to this information in AST form, then, it uses the mapping operations to yield the query $m^{-1}(uses(m(s)))$.

The PDG mappings provided by the AST-PDG mappings submodule are based solely on PDG variables, although they are not the only objects in the CFG that need to be accessed or mapped back to the AST. However, based on the context of the operation that needs the mapped data, one or more mapped variables can be translated into the object actually needed. For example, to find the definition of a function in the PDG, the AST function expression is mapped to a CFG variable definition, perhaps a temporary. By calling contained-in on the variable, the enclosing CFG statement can be reached, which contains the function definition in its argument field. The unified layer hides these mechanics in higher level mapping operations, simplifying the unified semantic queries supplied by the rest of this layer.

2) Generalizing the Syntactic Interface: How does this layer allow queries about program components that do not (yet) exist? One way is to minimize the dependence of a query's interface on the AST encoding of the construct under consideration. For instance, a transformation might be intended to extract a sequence of statements as a function definition and move it to a new location. The extraction requires identifying any variables used by the statements that will not be visible from the new function's location. However, as the statements exist in the AST, they are not yet a single composite AST data element (or even necessarily contiguous), as might normally be required by a query's interface. Rather than prematurely convert them into a function body to collect
the information, the query's interface is generalized—without loss of accuracy—to treat a list of statements as though it were the body of a function. This approach avoids unnecessary temporary AST transformations just for checks. The downside is that such queries must have extra code to handle both the special and the normal cases.

Another common case is planning to remove a construct from a composite construct, such as a statement from a compound statement. In such a case the properties of the composite construct after removal need to be computed. One way to handle this without transformation is to apply the query to the composite, then apply the query to the construct to be removed (perhaps in a modified form), and then use a set-difference operation on the two results to compute the desired result.

D. The Global Transformation Layer

The global transformation layer implements the checks and transformations of meaning-preserving AST transformations in terms of the consistent and unified abstraction provided by the lower layers. Checks and transformations are separated to provide greater flexibility at layers above (e.g., the restructuring layer) and to keep operations of manageable size.

Because consistency, translation between program abstractions, complex queries, and local changes are managed by the lower layers, the global transformation layer is simple in structure. However, a restructuring operation's checks and transformations are still complicated to implement because global program properties must be examined and manipulated to preserve meaning.

To provide some additional help, then, the globalization skeleton is used to structure the process of deriving a meaning-preserving AST transformation from the semantic queries and local changes provided by the unified layer. It encodes the basic restructuring paradigm: the software engineer using the tool applies a restructuring change to a single source-level component c, and the tool applies this and other global compensating changes to make the change meaning-preserving. In particular, the skeleton for defining a transformation mp-ast-trans is:

```
procedure mp-ast-trans(c)
  for u_i ∈ affected(c) do
    compensation-trans(u_i)
  initial-trans(c)
end
```

where initial-trans is the tool user's requested local change to c, affected retrieves the set of components that will be affected by the application of initial-trans to c, and compensation-trans is the change to each of those references. By making appropriate choices of initial-trans, affected, and compensation-trans and by combining them to change c and the affected components in concert, the meaning of the program can be preserved. Both initial-trans and compensation-trans are typically simple movements, copies, and substitutions, along with some deletion and creation of syntax to represent the new structure. An instantiated skeleton is shown in Section IV-F.

The semantic checks themselves are not specified by this skeleton, but are implied by the generality of the compensations relative to the initial change. In particular, the compensations designed by the transformation programmer may not be general enough to handle all possible initial changes. (Generality may be compromised because a compensating change is impossible or because the transformation programmer chose to simplify the implementation of the transformation.)

Interleaving queries and transformations can lead to difficulties, since a partially transformed program might be an incorrect program. For instance, if a variable name is being changed, first the name in the definition of the variable is changed, then all of its uses are likewise updated. Until a use is changed, it is "undefined". Queries during such an intermediate point could yield incorrect results or force a reconstruction of the PDG that would fail because the AST is semantically ill-formed. Consequently, it is conventional for a transformation to retrieve all the globally related elements before performing any changes to the AST.

E. The Restructuring Layer

The restructuring layer defines the complete meaning-preserving transformations by combining the transformations and checks of the global transformation layer, along with reports for check failures. In addition, this layer is responsible for the timely update of the PDG with respect to the AST.

As an example of the value of separating a meaning-preserving transformation's suboperations below the restructuring layer, consider the separation of a transformation's check and the report of failure. The check returns the result representing failure, but leaves the restructuring layer to report the details to the tool user. Consequently, it is possible for a transformation in the restructuring layer to use the failure to guide alternative attempts to transform the program as requested by the user. This separation has been exploited by a high-level graphical restructuring interface for encapsulating datatypes [32].

The global transformation layer knows primarily about changes to the AST and queries on that abstraction. However, after the AST is updated, the PDG must also be updated to reflect the AST's new structure. This is not a complex task, but it does acknowledge the existence of auxiliary abstractions. Consistency cannot be entirely managed by the AST, because some dependent abstractions may not be interested in—or behave correctly for—all intermediate states of the AST. If the PDG or other abstractions were updated after each change to the AST, not only would performance suffer, but errors might result when rebuilding the auxiliary abstractions; this is due to the same problems that could arise in the global transformation layer if queries and transformations were interleaved. Consequently, the restructuring layer is responsible for rebuilding the PDG at the correct moment—sometime before a query to the PDG is made. Additionally, the restructuring layer is responsible for registering the consistency submodule with the AST's change events, since registration is closely tied to timely update.

F. Implementing var-to-expr

To give a sense of how the submodules depend on each other, particularly with respect to the layering, the task of
implementing var-to-expr is described. The presentation is given in terms of the basic tasks the transformation programmer carries out, with commentary on implementing var-to-expr in particular.

The transformation var-to-expr is used because it is one of the most basic meaning-preserving restructuring transformations; it replaces a designated variable use with the expression that defines it. The discussion is simplified by assuming that the selected expression is a simple variable definition (not a variable use or a binding declaration) and by eliminating some flexibility options.

The task is presented bottom-up, starting in the unified layer’s AST submodule. Since most operations required of the unified layer are implemented already, most of the implementation task is actually focused in the global transformation and restructuring layers.

1) Choose the local transformation that the tool user logically applies. For var-to-expr the variable carrying the value of an expression is being deleted. This effect is achieved with the operation ast-remove!, which is defined in the unified layer in the AST submodule.

2) Choose the transformation that must compensate the local change. Deleting a variable definition is compensated by inserting the defining expression in all the places where the variable definition is used. The compensation is the low-level operation ast-subst!, which again is defined in the unified layer.

3) Formulate the queries that select the program components to be compensated. To retrieve the uses of the variable being deleted, get-uses needs to be called, which is defined in the unified layer.

4) Implement a transformation combining the local and compensating transformations. This transformation is modeled after the globalization skeleton. The instantiation of the globalization skeleton for var-to-expr looks like:

\[
\text{procedure var-to-expr}(\{v := e\}) \\
\text{for } u_i \in \text{get-uses}(\{v := e\}) \text{ do} \\
\text{ast-subst}!(\text{ast-copy}(e), u_i) \\
\text{ast-remove!}(\{v := e\})
\]

where \(\{v := e\}\) is the expression that defines \(v\). It takes the variable \(v\) of the selected expression, finds its uses through get-uses, removes the expression with ast-remove!, and then calls ast-subst! to replace the uses with a copy of the expression that originally defined the variable.

5) Choose the checks for the top-level transformation. A check generally examines the semantic dependents of the changing object, and removes those that are being compensated and those that are unaffected. If there is anything left, the transformation’s compensations will not be sufficient to preserve meaning.

There are several checks for var-to-expr. First, the expression to be inlined must in fact be a variable definition (for the simplified example). Second, moving the expression must not change its meaning. This requires calling move-expr-check-semantics—defined in the global transformation layer—on the defining expression with respect to each variable use returned by get-uses. Third, multiple evaluations cannot have side-effects. Thus if the list returned from get-uses has length greater than one, a false value is required from side-effects? on the expression. Last, none of the uses to be replaced can have two possible definitions (for example, one each in the arms of a conditional). This is checked with no-multiple-defs-check, also defined in the global transformation layer, which calls get-definitions (the converse of get-uses) on each variable use to make sure there is only one. The checks are put into var-to-expr-check.

6) Combine the top-level check and the transformation into a meaning-preserving transformation. This is a simple step, and the code looks something like:

\[
\text{procedure checking-var-to-expr(var)} \\
\text{signal-structure-start()} \\
\text{if var-to-expr-check(var) then} \\
\text{var-to-expr(var)} \\
\text{else} \\
\text{print-failures()} \\
\text{signal-structure-complete()} \\
\text{end}
\]

The signal-structure-start call makes sure that the program abstractions are consistent and the PDG is ready to be accessed, and prevents the PDG from being updated on every change to the AST in the subsequent calls. The signal-structure-complete call marks the end of the action, allowing the PDG to be updated.

Defining a new transformation is often relatively straightforward, since the lower-level tasks (for instance, checking for multiple definitions) are common to many transformations and are already provided by the unified layer to the global transformation layer (or by the global transformation layer itself). This is especially true for the syntactic transformations, and the basic semantic queries and checks. Also, since the basic mechanisms of consistency are in place and queries are put before updates, the calls on the AST and PDG are straightforward. The major difficulty is in correctly stating checks, which requires knowledge of the semantics of the language between restructured as well as the semantic relationship between the old and new structure.

G. Discussion

The AST, PDG, and mapping modules are each encapsulated using Common Lisp’s package construct. A module is typically divided into files such that each file is a submodule. No explicit import-export mechanism is used to control access to submodule or layer interfaces. The tool’s hierarchical type structure is described by CLOS classes, and
methods implement units of function. An implementation of Sullivan’s abstract behavior types (ABT’s) [9], [10], [33] is used to propagate the effects of changes on the AST to other abstractions.

The initial tool was implemented in a bottom-up fashion following the basic architecture. Typically, code was added to a lower layer only when it was needed by a higher layer, permitting the higher layers to be reached relatively early in development. The basic architecture isolated one key issue in each layer, simplifying the design and implementation by allowing the tool builder to focus on just one issue for each programming task. The column structure further separated the tasks within a layer according to the basic program abstractions: The implementors of the AST and PDG abstractions were separated below the global transformation layer, with the mapping module providing the required consistency and mappings between the data elements of the abstractions. Because changes to the AST were propagated with event announcements rather than direct calls, unforeseen changes to the mapping module’s implementation of the consistency mechanism did not affect the AST’s implementation. The unification of program abstractions through the AST hid the PDG from upper layers, simplifying the programming of queries and transformations. Additionally, changes to the PDG implementation or interfaces would not affect the global transformation or restructuring layers. Because the unified layer is generic, it can support extension to tasks other than restructuring. For instance, a program slicer and other visualizations have been added to the tool with little effort.

It was not hard to maintain the basic integrity of the architecture during this first phase. It was relatively easy to distinguish AST code and PDG code, and there was little reason to violate module or layer barriers. Typically, a module barrier was violated for reasons of convenience. However, later it was often discovered to result in redundant low-level code, decreasing maintainability. Most of the harmful cases have been abstracted as higher level functions and imported through the mapping module. Layering violations are infrequent because they usually require writing extra code to provide the function of a circumvented layer. However, because common lisp lacks support for layers, these violations have been harder to detect. Osher’s grids [19] would be useful in capturing layered and modular structure together.

One small problem was that it was not always clear what was AST code and what was mapping code. In particular, although the low-level mapping functions map from a CFG variable to an AST expression, extra code is required to combine a group of such low-level mappings into a single AST expression. This combination step uses only AST operations, so it appears that it might belong in the AST module. However, this combining step subtly depends upon what the mappings mean, and the code actually belongs in the mapping module’s flow query submodule. Because there are no syntactic hints of a violation (e.g., use of mapping operations in the combining step), some of these violations can remain hidden in the AST module. Such violations did not occur in the PDG module because—at this point—it was known that the entire PDG module should reside below flow query.

V. THE EXTENDED ARCHITECTURE

The basic architecture provides a powerful basis for manipulating complex data abstractions—in our case for performing meaning-preserving transformations. However, the expressiveness and control added with each successive layer makes certain goals harder or impossible to meet [4]. Parnas noted:

As one goes higher in the levels, one can lose capabilities . . . not gain them. On the other hand, at the higher levels the new functions can be implemented with simpler programs because of the additional programs that can be used. We speak of “convenience” to make it clear that one could implement any functions on a lower level, but the availability of the additional programs at the higher level is useful [3, p. 314].

In the case of our basic architecture, this balance led us to isolate all but one aspect of AST–PDG consistency from the restructuring layer; basically, the only consistency knowledge required of the restructuring layer is about when the PDG must be consistent. Such limited knowledge simplifies using the global transformation layer. However, in the process, the capability to precisely control consistency has been lost.

After constructing the basic architecture and a collection of restructuring operations, it became apparent that this decision had been costly because a PDG is expensive to construct from scratch after each restructuring transformation completes. The expense is due primarily to the pointer alias information that a PDG maintains to reason about the dependencies between program components; computing alias information can require between $O(N)$ and $O(N^3)$ time in the number of variable references in the program, depending on the algorithm chosen [15], [29], [34]. Performance supporting interactive restructuring, on the other hand, requires closer to linear time with respect to the number of changes to the AST.

Because the upper layers perform meaning-preserving transformations, the use of the lower layers is quite stylized (as characterized by the globalization skeleton). This quality suggests that the changes to the PDG might be analogously stylistic. However, the separation provided by layering, as well as the desire to preserve simplicity and generality of the lower layers, generally discourages the implementations of lower layers from exploiting the properties of the upper layers. Consequently, in the basic architecture the layers below the global transformation layer are unaware of this stylized use. To achieve the desired performance without violating layering or modularity, we considered two solutions. The first approach, generic incremental update [26], [28], [35], records local changes to the AST and then reperforms data flow analysis on all portions of the program affected by the changes. This approach would leave the basic architecture unchanged, since it can be implemented entirely within the consistency layer, but it comes at the expense of complicating the consistency layer. However, there is little evidence about how well the generic update approach would perform in the context of restructuring operations as described in the previous section. Additionally, our tool requires general alias analysis,
which is not yet supported with generic incremental update approaches [28], [29].

A. Direct Update

The second approach, direct update, takes into account exactly what operation is being performed on the program to update the PDG information [27]. We took this approach because it can handle general aliasing and because it showed greater potential for improving performance of restructuring than did generic incremental update. It, too, however, comes at the cost of added complexity.

The direct update approach takes advantage of the aggregate effects of the operation to implement a more succinct update to the data flow information; with sufficient knowledge, the cost of the update can be proportional to the number of changes to the AST.\(^4\) On the other hand, this technique is application-specific and requires that a tool’s individual operations be semantically meaningful or that some meaning can be derived from a combination of them during use. The program restructuring tool has such operations: each transformation guarantees to preserve the input–output behavior of the program. Moreover, a transformation’s update is stylized in structure.

A direct update algorithm can be modeled as the application of two equivalent functions to two different representations, one for the program text \(p\) and one for the data flow information \(d\):

\[
[F_{ast}(p_{old}), F_{pdg}(d_{old})] \rightarrow (p_{new}, d_{new}).
\]

The updates to mappings can be modeled similarly as \(F_{map}\). Although conceptually simple, the direct update approach is complicated because each operation \(F_{ast}\) must be translated by hand and implemented as an operation \(F_{pdg}\). Also, the basic tool architecture must be changed to incorporate \(F_{pdg}\) because it logically belongs in the global transformation layer with \(F_{ast}\) where no PDG operations previously existed.

To help the transformation programmer construct an \(F_{pdg}\) for an \(F_{ast}\), the globalization skeleton for a transformation (see Section IV-D) is used in conjunction with a set of PDG subgraph substitution rules, which describe meaning-preserving changes to the PDG that can be used to mirror the restructuring changes to the AST. First, the globalization skeleton is used to identify the PDG nodes and edges of interest for the update. The nodes are mapped from the AST objects \(c\) and \(u\) denoted in the skeleton. The primary edges of interest are the ones traversed by the affected function, so the core of this function can be used to retrieve the edges. The transformation programmer then examines how the AST transformation mutates the affected relation, and determines the analogous PDG change in terms of the mapped PDG nodes and the PDG relation underlying affected. The transformation programmer then matches these changes to a composition of PDG substitution rules.

Unlike the globalization skeleton, which decomposes an AST transformation according to the syntactic separation of its changes, the PDG substitution rules divide the work according to structural and semantic responsibilities. For instance, there

\(^4\)The approach of looking at the operations performed rather than the changes in the data is not unlike Lippe and van Oosterom’s operation-based technique for merging program versions [36].

are different rules for changing data flow dependencies and control dependencies. When applied together to describe a single complete update, their substitutions change different aspects of the same subgraph. This approach to dividing a complex PDG update into smaller pieces eases understanding because the semantic integrity of each piece is guaranteed. On the other hand, this division means the PDG has no corresponding AST during the PDG substitutions for a restructuring change. This has no negative impact, however, because the AST’s and PDG’s updates are performed independently.

The substitution rules also specify preconditions for their correct application. These preconditions require semantic checks to be performed before a tool transformation is applied. (The substitution rules are also useful for determining if an AST transformation is correct.)

One PDG substitution rule is the distributivity rule, which is used, for example, to replace the portion of the PDG for a variable use with a copy of the PDG for the expression that defines it (see Fig. 5), or vice versa. Intuitively, the rule states that if an expression’s result is assigned to a variable, then a copy of that expression may replace a subsequent reference to that variable definition. Because replicating the expression is essentially multiple evaluation of it, the side-effects that it is allowed to have are restricted; the implementation of the distributive rule must check to make sure that this restriction is satisfied. There is also a transitivity rule for exploiting the transitivity of assignments, and a control rule for changing the control dependence of moved code.

B. Adding Direct Update

The basic architecture must be changed to incorporate \(F_{pdg}\), since it naturally belongs in the global transformation layer with \(F_{ast}\), but no PDG operations are exported beyond the consistency layer. Since the tool is constructed in layers, and because the AST and PDG submodules of those layers are relatively independent, a natural approach to resolving this problem is to augment the existing architecture in the PDG module (i.e., column) by adding additional submodules at the higher layers. The mapping module must also be extended with submodules at these layers to permit incrementally updating the AST–PDG mappings.

For the PDG module, the substitution rules guide the evolution to the new architecture, in particular deciding how the new function should be divided into layers.

1) Extend the consistency layer. The consistency layer must now export PDG update operations to PDG submodules at higher layers. Likewise, mapping update operations must be exported to the unified layer. Because the repre-
sentations are sometimes updated a little differently than the way they are originally constructed, some additional operations have to be implemented. For instance, some mappings are created by inserting into a list. Updating a mapping, however, can require clearing the original mapping first, which was not a required operation in the basic architecture, and hence not implemented.

2) **Extend the unified layer with substitution rules and updates to mappings.**

   a) **Add the substitution rules.** Applying one substitution rule makes sense in the PDG, but the result does not necessarily correspond to a legal AST. In this sense these rules are analogous to the AST local update operations of the unified layer, of which several must be applied to yield a program that is representable as a PDG (not to mention preserve meaning). Consequently, the substitution rule implementations are placed at the unified layer of the PDG column.

   b) **Add mapping updates for the substitution rules.** Extending the mapping module to accommodate the incremental changes of the substitution rules uses the newly exported primitive mapping update operations. For example, the mapping module must perform the following in conjunction with an application of the distributivity rule:
   
   - **Delete the removed variable's mappings.** Clear the mappings related to the deleted AST variable.
   - **Map copied expression.** Create the maps for each copied AST expression to its associated variable in the CFG. The map is isomorphic to the maps for the original expression and its associated CFG variables.

3) **Add the F_{pdg} and F_{map} operations at the global transformation layer.** F_{pdg} and F_{map} naturally reside at the same layer as F_{ast}, since they are isomorphic with respect to their underlying representations. However, each resides in its own independent submodule.

4) **Augment the restructuring layer.** Changes are made to the restructuring layer to call the corresponding F_{pdg} and F_{map} after F_{ast}, passing the information required to make the correct update. This augmentation can also be accomplished as a pure extension by using CLOS’s auxiliary method mechanism. This approach also achieves the integration one layer lower—at the global transformation layer—easing enhancement in the restructuring layer.

This approach not only allows extending the architecture in layers, but also permits first implementing a transformation in “batch mode,” and then later implementing the direct update for it. Consequently, the separation of correctness concerns versus performance concerns persists in the extended architecture.

**C. Implementing var-to-expr**

Implementing a unique F_{pdg} for each F_{ast} is an extra responsibility on the transformation programmer. However, because the mapping and PDG modules are independent, the extra requirements do not affect the original coding of the AST transformation. In addition to the steps listed in Section IV-F, the transformation programmer must now perform the following steps.

1) **Map F_{ast} to PDG substitution rules.** AST transformation var-to-expr maps to the distributive rule to replicate the portion of the PDG representing the expression and replace the variable use with it, and the control rule to change the location (i.e., control dependence) of the replicated expression to that of the replaced variable use. If a transformation cannot be successfully mapped, then either a new rule is added (and implemented in the unified layer) or the AST transformation is reformulated to stay within the requirements of direct update.

2) **Implement F_{pdg} at the global transformation layer.** The programmer combines the selected PDG substitution rules into a complete transformation. For pdg-var-to-expr, this is straightforward, except that an extra application of the transitivity rule is required to keep the CFG statements in normal form (see Section IV-B).

3) **Implement the mapping update, F_{map}**. Any changes to AST–PDG variable associations outside the scope of the substitution rules must be implemented. For pdg-var-to-expr, the deleted variable references must be unmapped, and the inclined expressions might change expression mappings for the expressions that contain them. Recall that mapping was an insignificant issue when implementing a transformation in the basic architecture.

4) **Change the restructuring layer’s call of the transformation to call F_{pdg} and F_{map}, passing each the information needed from F_{ast}.** Function checking-var-to-expr now looks like:

   - procedure checking-var-to-expr (var)
     - signal-restructure-start()
     - if var-to-expr-check(var) then
       - es-and-uses := var-to-expr(var)
     - changed
     - pdg-es := pdg-var-to-expr(var, es-n-uses) # added
     - map-var-to-expr(var, es-n-uses)
     - pdg-es # added
     - else
       - print-failures()
       - signal-restructure-complete()
   - end

In some cases, like the above, the AST transformation has to be modified to return the correct information, but this is not a disruptive change because it is only changing return values; until the architecture was extended, few calls used the return values of transformations.

**D. Discussion**

The basic architecture proved robust with respect to a complicated change. In particular, the architecture was extendable in a systematic way to address a serious performance problem. The resulting implementation reduced the cost of reconstructing the PDG from polynomial in the size of the AST to linear in the number of updates to the AST, typically
a few seconds [27]. In contrast, the batch updates in the basic architecture could take hundreds of seconds on a modest program. In spite of this dramatic improvement, the benefits of the basic architecture persisted in the new architecture. The added code constitutes about 8% of the overall system.

However, this new architecture comes at some cost. First, the transformation programmer must implement \( F_{pdg} \) in the global transformation layer. Also, the consistency layer must export PDG and mapping update operations to the unified layer, in addition to the query operations it provided before. Likewise, the unified layer must also export the PDG substitution rule operations and corresponding mapping updates to the global transformation layer for implementing \( F_{pdg} \). These extra operations have two consequences. One, the unified program representation layer is no longer uniformly AST-oriented, and deserves a new name to reflect its wider interface. Two, the interface to the PDG voids the guarantee of consistency with the AST and the AST–PDG mappings, adding to the complexity of formulating meaning-preserving updates to the AST. The tradeoffs are obvious: good performance comes at the expense of exposing a more complicated and dangerous interface.

These problems are somewhat mitigated by the column structure imposed by the architecture, especially at the extended layers. Transformations on the AST and PDG are kept separate below the restructuring layer by the mapping module, and PDG operations do not access the AST. Consequently, calls made below the restructuring layer in Fig. 3 primarily flow straight downward. The only left downward calls are due to the mapping module’s mapping of PDG elements to AST elements. The right downward calls from the AST through the mapping module to the PDG perform no updates, except when loading a new program. This additional structure is helpful because the restricted call structure simplifies reasoning about the relationships between layers, as well as between modules. The independence of the AST and PDG modules minimizes implementation changes in one module from propagating into the other. Although many of these benefits were present in the basic architecture, they were not especially evident until the PDG and mapping columns were extended up to the restructuring layer.

On the other hand, the direct updates are designed specifically for restructuring, so building new applications on top of the program representation layer would not allow reusing the upper PDG submodules. However, it is not surprising that removing one aspect of a highly specialized layer (the AST submodule of global transformation) precludes reusing the other submodules of the layer. On the other hand, the extended architecture does not compromise the generality of the lower layers, so they can be reused.

VI. CONCLUSION

Layered architectures can be found in many domains, ranging from operating systems and network protocols to tools that manipulate complex program representations. Despite this widespread use and despite general knowledge about layered architectures, our discipline still falls short of making the design, use, and evolution of layered architectures a routine matter of engineering. There are several contributions that this paper makes in pursuit of this objective.

- We provide a prescriptive approach to constructing layered systems in which information hiding modules are distinct entities. Layers—"horizontal" abstraction—are defined according to specific properties that they maintain. For example, the consistency layer ensures that the AST, the PDG, and the AST–PDG mappings represent the same program source. Modules—"vertical," columnar abstraction—are defined to manipulate a given entity (for example, the AST, the PDG, or the AST–PDG mappings). Submodules are the parts of a module associated with a given layer.
- We show how separating the implementation of mapping between modules into its own module—in the style of mediators [9], [10]—improves the ability to reduce the complexity of interactions both between and within layers, as well as between modules.
- We empirically confirmed the potential for poor performance in layered systems. In our base architecture, we identified the central reason for this to be because lower layers did not take advantage of properties of higher layers. In particular, the desire to preserve the simplicity, generality, and independence of the lower layers discouraged exploiting the stylistic use of the global transformation layer by the restructuring layer. The desire to preserve modularity exacerbated this problem.
- We presented a disciplined approach to overcoming this performance problem by exploiting the layer and modular abstractions of the architecture. Specifically, we redefined, in an incremental way, the specific properties that layers provide. Then we extended some of the modules upwards (i.e., we exported hidden submodule interfaces and added submodules in upper layers), allowing the highest layers to apply more precise operations on those modules. Finally, we extended, in a careful and consistent way, the topmost layer to take advantage of these newly provided operations. In essence, it was possible to separately consider the implementation of a correctly functioning tool from the issues of performance.

In addition to our own experience, this approach has been successful in helping to manage several issues in the development of a program understanding and transformation tool [11]. To give one example, paging overhead can cause a severe performance problem in such a tool due to the space occupied by the AST. However, if the AST is being processed in a single pass, space usage can be optimized by constructing, processing, and discarding small portions of the AST during parsing of the program text input. By treating the parser as a mapping column between the text and the AST representations, it was possible to systematically extend the tool to support such special processing without compromising the AST abstraction.

ACKNOWLEDGMENT

The authors would like to thank J. Larus for providing us with the Curare system. We are grateful to K. Sullivan for his guidance in using mediators to integrate the AST and PDG. We are thankful to the referees for their insightful comments.
and to G. Murphy and H. Osbør for their help in clarifying our ideas. We thank D. Garlan for his help with clarifying our ideas on modules in a late draft of this paper. Much of this work was inspired by N. Habermann’s technical contributions and by his dedication to software engineering research and education.

REFERENCES


William G. Griswold (S’89–M’91) received the B.A. degree in mathematics from the University of Arizona in 1985, and the Ph.D. degree in computer science from the University of Washington in 1991. He is an Assistant Professor in the Department of Computer Science and Engineering at the University of California, San Diego. His research interests include software evolution and design, compiler technology, programming languages, and parallel computing.

Dr. Griswold is a member of the program committee for the 17th International Conference on Software Engineering, and is Cochair of the Workshop on Program Transformation for Software Evolution at the same conference.

David Notkin (S’77–M’82–F’83) received the Sc.B. from Brown University in 1977, and the Ph.D. degree from Carnegie-Mellon University in 1984. He is a Professor in the Department of Computer Science and Engineering at the University of Washington. He joined the faculty in 1984, after receiving the Ph.D. degree.

Dr. Notkin received a National Science Foundation Presidential Young Investigator Award in 1988. He was a visiting faculty member at Tokyo Institute of Technology and at Osaka University in 1990–1991. He was Program Chair of the ACM SIGSOFT ’93 Symposium on the Foundations of Software Engineering and is Program Cochair of the 17th International Conference on Software Engineering. He is serving as the chair of the Steering Committee for the International Conference on Software Engineering, and also serves as the Treasurer for ACM SIGSOFT.