Component Design of Retargetable Program Analysis Tools that Reuse Intermediate Representations[*]

James Hayes[+] **William G. Griswold**[+] **Stuart Moskovics**[†]

[+]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114 USA
{wgg,jyuan}@cs.ucsd.edu

[†]Motorola, BCS
6450 Sequence Drive
San Diego, CA 92121 USA
smoskovics@gi.com

ABSTRACT
Interactive program analysis tools are often tailored to one particular representation of programs, making adaptation to a new language costly. One way to ease adaptability is to introduce an intermediate abstraction—an adaptation layer—between an existing language representation and the program analysis tool. This adaptation layer translates the tool's queries into queries on the particular representation.

Our experiments with this approach on the StarTool program analysis tool resulted in low-cost retargets for C, Tcl/Tk, and Ada. Required adjustments to the approach, however, led to insights for improving a client's retargetability. First, retargeting was eased by having our tool import a tool-centric (i.e., client-centric) interface rather than a general-purpose, language-neutral representation interface. Second, our adaptation layer exports two interfaces, a representation interface supporting queries on the represented program and a language interface that the client queries to configure itself suitably for the given language. Straightforward object-oriented extensions enhance reuse and ease the development of multi-language tools.

Keywords
Retargetability, reuse, software design, program analysis, software tools.

1 INTRODUCTION

By summarizing information that is collected from a software system as a whole, a program analysis tool can reduce the time required by a programmer to understand a system well enough to begin making changes. For example, RIGI can summarize the architectural elements of a system [10]; our StarTool provides hierarchical, crosscutting views of all the uses of a data structure or other design decision [7].

Such tools are most useful if they are able to analyze programs written in varied source languages, since systems that benefit the most from such analysis, especially legacy programs, are often written in multiple or proprietary programming languages. However, program analysis tools are often tailored to one particular representation of programs, making adaptation to a new language costly (Figure 1a). An inexpensive way of achieving adaptability is to introduce an intermediate abstraction—an adapter component or adaptation layer—between an existing language representation and the program analysis. This additional layer translates the tool's queries into queries on the particular representation [4, pp. 139–150]. Genoa [2] represents such an approach for accommodating multiple analysis tools (See Section 6).

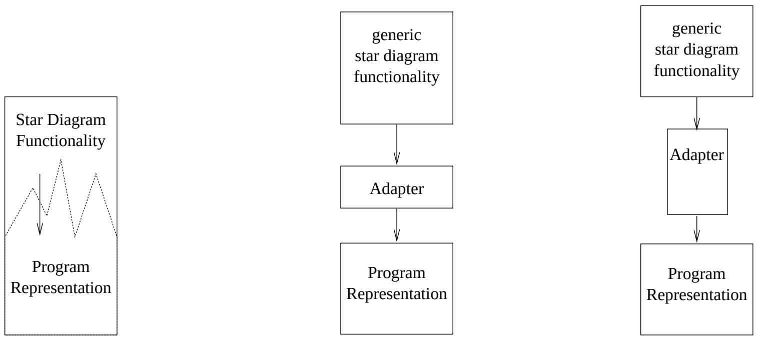
The question arises, then, as to what interface should lie between the adaptation layer and the retargeting tool. An appropriate interface would impose minimal inconvenience on adaptation layer implementations, since one must be written for each retarget to a language representation.

In a project to make our StarTool program analysis tool easily retargetable, we hypothesized that the adaptation layer interface should be a low-level, language-neutral, tree-oriented language representation interface, because it would impose minimal responsibilities and inconvenience on any representation (Figure 1b). Only a small number of operations should be required because of StarTool's limited needs.

To assess this claim, we first restructured StarTool to remove representation- and language-specific references in the code and have it import such an interface in their place. We then developed retargets to the Ponder C program representation [5], a similar representation for Tcl/Tk, and the Gnat Ada [3] program representation.

Although these representations are indeed tree-based, the export of a low-level and language-neutral interface complicated the adaptation code and hurt performance because it made inappropriate assumptions about the language representation. It also prevented language-specific traits from being expressed in the tool's user interface. Our mistake was to treat the design of the adaptation layer interface like the design of a service that is reused by many clients, when in our case the component being reused is an incomplete client that could import a number of possible services. Hence, the effort

*This work supported in part by UC MICRO grants 97-061 and 98-054 joint with Raytheon Systems Company, and NSF Grant CCR-9508745.



Star Diagram Functionality
Program Representation
generic star diagram functionality
Adapter
Program Representation
generic star diagram functionality
Adapter
Program Representation

(a) C-only version of StarTool (b) Representation-centric design (c) Tool-centric design

Figure 1: Evolution of StarTool's design for retargetability. Edges denote the *uses* relation, realized by function call. The shape of the adapter box reflects the scope of its responsibilities: greater width implies supporting more functionality, greater height implies bridging a larger semantic gap.

in supporting a flexible, low-level interface was misplaced: the client did not need it, and so the adaptations did not need to support it. The implication was that our focus should be on designing an appropriate *requires* interface for the client that states only what the tool requires in terms of the its own features. This change in orientation widens the semantic gap between the tool and possible representations to give adapters increased implementation flexibility (Figure 1c). Language-specific configuration of the tool is achieved through interface operations that query the adaptation layer about how the tool should behave for language-sensitive tool features.

This choice requires the adaptation code to explicitly understand the relationship between the language representation and the tool's features. Yet this choice resulted in adaptations that are simpler and more efficient because it provides flexibility to the adaptation implementation. It also supports multi-language analysis through the introduction of an adaptation layer that joins two other adaptation layers: the multi-language adapter is concerned only with a join that meets the tool's particular needs rather than in general terms of what a multi-language program means. In short, an adaptation layer functions as a glue-hiding mediator that serves to combine independently developed components [13, 14].

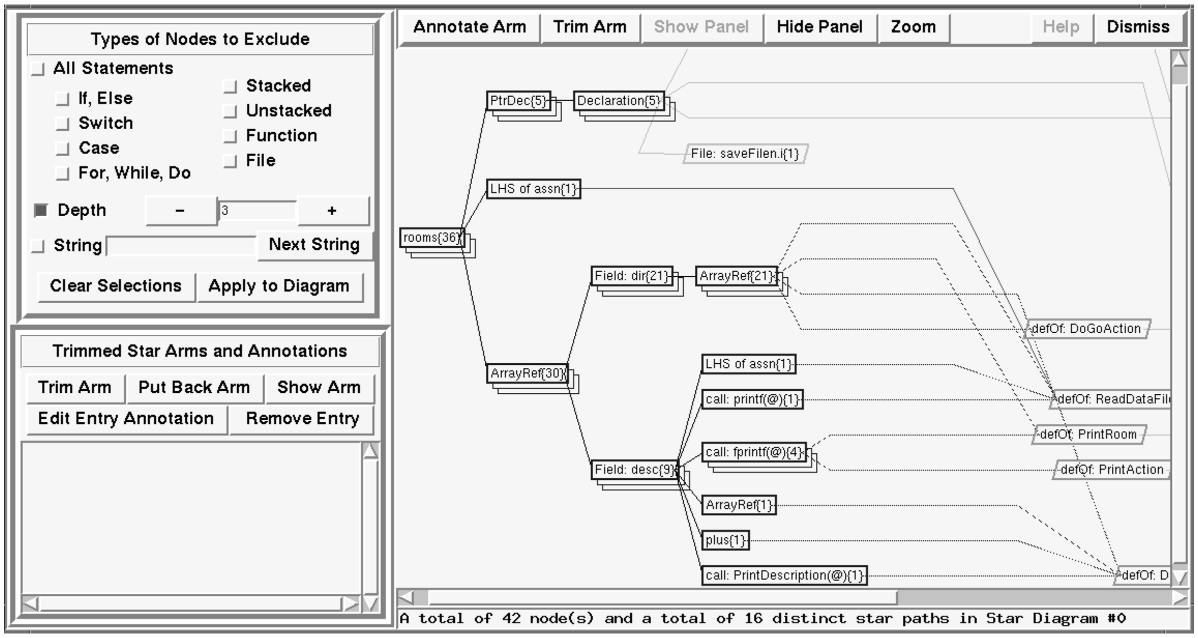
Section 2 describes StarTool and its representation requirements. Section 3 describes our initial retargetable interface, and Section 4 describes the Ponder C, Tcl/Tk, and Gnat Ada retargets, as well as the problems encountered. In Section 5 we discuss our insights, the resulting revised adaptation interface, and techniques for improving reuse and supporting multi-language retargets. We close with consideration of related retargeting approaches, a design process for designing client-centric interfaces, and criteria for application to other program analysis tools and software.

2 STARTOOL AND ITS REPRESENTATION NEEDS

The StarTool program analysis tool assists programmers in planning restructuring projects on large software systems [7]. In particular, it helps a programmer to make encapsulation decisions by displaying context graphs called *star diagrams* that provide information about the usage patterns of objects within a program. To build a star diagram, the user selects objects of interest, typically variables, and the tool expands these selections to include all references to the selected objects, or if requested, all references to objects that have the same type as those objects.

Figure 2 shows a star diagram for the variable `rooms` built for a program consisting of four C source files. The root of the diagram (the leftmost node in the graph) contains all references to the variable itself. The children of the root node are the syntactic constructs that contain the root references; children designating identical syntactic constructs, such as the 30 array references, are stacked into a single node in the display. Similarly, the grandchildren of the root node are the syntactic constructs that embed the child nodes, and so on, out to the leaves of the diagram, which represent source files. In order to reduce clutter in the display and convey context information, functions and files are represented by singleton nodes in the display, rather than being replicated.

Since the stacked nodes in a star diagram represent multiple, identical uses of an object, they point out the most likely candidates for abstract operations on the object in a restructured system. The tool supports exploration by allowing the user to inspect the program text associated with a node. Planning a restructuring is supported by allowing the user to trim a path from the star diagram display into the lower left-hand panel and attach descriptive text to it. After planning is complete, the trims can be viewed in their own star diagram views and used to navigate back to the program text to carry out the restructuring change.



```
A total of 42 node(s) and a total of 16 distinct star paths in Star Diagram #0
```

Figure 2: A star diagram window for the variable `rooms`.

Because star diagrams can contain thousands of nodes, the tool allows the user to remove uninteresting nodes from the display. The panel in the upper left-hand corner of the star diagram window presents the user with a selection of node attributes; nodes possessing any attribute selected by the user are elided, and connections joining the node to a parent are redirected to the node's children.

There are three major stages to the construction of a star diagram from a hierarchical program representation such as an abstract syntax tree (AST) [1]. The first stage is construction of a root set. The objects selected as the prototypes of the root set are expanded into the full root set by traversing the AST and testing each AST node to see if it should be included according to the user's chosen similarity criterion, either same variable or same type. The second stage computes the descendants of the star diagram root by retrieving the parents of the AST nodes of the root set and classifying them according to how they manipulate the child. Concretely, this is achieved by comparing the labels that the AST nodes generate and clustering those that have the same label. This stage is repeated on each node in the star diagram until only file nodes, which constitute the leaves of the star diagram, remain. The third stage computes a graphical layout of the star diagram, including the merging of the function and file nodes.

3 RESTRUCTURING AND INITIAL INTERFACE

Given the relatively focused needs of StarTool—basic tree traversal capabilities and some semantic comparisons—we hypothesized that only a small number of relatively simple operations would need to be exported by an adaptation layer to successfully separate StarTool from underlying program representations. Based on our experience in designing and using ASTs [5, 8], we felt that a low-level, language-neutral interface for the adaptation layer would be least troubling in writing adaptations that would ultimately span many languages. In particular, we felt that providing low-level functions would free adaptations of responsibilities to provide high-level functionality, and lack of language bias would simplify an adaptation that did not fit such a bias.

To minimize assumptions about what an adaptation could implement, we planned to limit the interface to using a few simple types. The interface would communicate AST information in terms of integer-sized opaque AST-node references. Except for testing simple equality, the only legal operations on these references would be provided by the adaptation module interface. Other data values, like positional information or file names, would be represented with least-common-denominator types such as integers and strings.

Isolating program representation details with an intervening adaptation module required restructuring [8] higher-layer routines into representation-dependent and representation-independent pieces, then rewriting the representation-dependent code using the services defined by our adaptation module. We attempted to retain as much of the program analysis and other algorithmic components in the higher layers of the tool as possible, moving only the aspects of AST manipulation into the new module. This served our long-term goal of easing the adaptation of the tool to a new program representation by limiting the complexity of the functions in the adaptation module.

To achieve representation independence and generically accommodate language constructs that could have any number of children, we chose an interface similar to that supported by Ponder [5]: the children of an AST node are accessed by a leftmost-child operation and successive right-sibling operations starting at the leftmost child; another operation provides access to a node's immediate parent. The traversal functions, as shown near the top of Figure 3, are `ast_child`, `ast_sibling`, and `ast_parent`.

To provide the semantic queries on the representation needed for the construction of meaningful star diagrams, a small set of generic query functions were chosen. These compare two nodes for variable or type equality (`similar`), advise which AST nodes should never be considered for inclusion in a star diagram (`ast_skip_test`), and provide a string representation of an AST node (`ast_label`). The function `ast_label_child_character` conveys which character is used to represent children in the labels; requiring every adaptation to use one particular character might complicate label production for some adaptation.

Since the programmer works with program text as well as star diagrams, functions are required for mapping between AST nodes and displayed program text. The functions `file_AstNode` and `find_AstNode` provide the capability to map from a file name or a text selection to an AST node, respectively. The functions `ast_file`, `ast_begins`, `ast_ends` convey from which file an AST node is derived and the text positions where an AST sub-tree begins and ends, respectively. Function `file_text` returns the text associated with a file.

Finally, function `ast_elaborate`, appearing at the top of Figure 3, permits an adaptation to do representation-specific initialization.

4 C, TCL/TK, AND ADA RETARGETS

C Ponder. As a first test of the new adaptation interface, we retargeted to C program ASTs generated by the Ponder language toolkit. Ponder combines a yacc-like grammar specification tool with data structure libraries and facilities for manipulating generic program ASTs and symbol tables [5]. The C instantiation of Ponder existed before we began the project [6].

The retarget resulted in 1000 lines of non-blank, non-comment adaptation code for the 14 functions. The size of the adaptation is larger than we expected in large part because about 250 lines of code were required to implement accurate and efficient AST–text mappings. The Ponder C implementation provides only file and line number mappings for AST nodes.

Tcl/Tk. We next moved to the issue of evaluating the interface for retargetability with respect to language issues, which were not touched upon in the C Ponder retarget since the

```
/*
** Performs any actions necessary to ready the
** module for use.  The parameters are those
** specified on invocation of the tool.
*/
int ast_elaborate(int &argc, char *argv[]);

/*
** Relatives of an AstNode in its tree.
*/
AstNode ast_child(AstNode item);
AstNode ast_parent(AstNode item);
AstNode ast_sibling(AstNode item);

/*
** Indicates whether two nodes are "similar".
*/
enum SimilarityTypes {SAME_SYMBOL, SAME_TYPE};
int  similar(AstNode left, AstNode right,
             SimilarityTypes similarity);

/*
** Returns a label representing the node.  This
** may contain occurrences of
** ast_label_child_character followed by a number,
** which the tool replaces to show where the child
** from whence this node was reached appears.
*/
char  *ast_label(AstNode item);
char   ast_label_child_character();

/*
** Returns True iff the node may be ignored for
** the purposes of constructing a star diagram;
** it's syntactic fluff we can ignore.
*/
int ast_skip_test(AstNode item);


/*
** Not quite a least-common-denominator type, but
** relatively simple, natural, and it avoids
** requiring two calls to acquire position
** information.
*/
struct FilePosition {
  int line, column;
};

/*
** Functions that perform AST-text/file mapping.
*/
char        *file_text(AstNode item);
char        *ast_file(AstNode item);
AstNode      file_AstNode(char *pathname);
FilePosition ast_begins(AstNode item);
FilePosition ast_ends(AstNode item);
AstNode      find_AstNode(AstNode start_node,
                          FilePosition start,
                          FilePosition end);
```

Figure 3: The initial adaptation module interface, which contains 15 functions.

original StarTool implementation was for C. We chose to retarget the tool to support programs written in the rather different Tcl/Tk language [11]. The representation for Tcl/Tk programs itself was designed to minimize representation adaptation issues, permitting a focus on language issues. The Tcl/Tk parser, AST generator, and text–AST mappings together consist of about 550 lines of code, plus a yacc grammar file of about 150 lines. We completed the retarget itself in about a week and under 300 lines of adaptation code, but encountered two problems in the process.

The first problem we encountered concerned the elision facilities of star diagrams. StarTool allows elision of case statement, loop statement (*do*, *while*, and *for*), and *if* statement nodes, as well as file and function nodes. Although Tcl/Tk includes these constructs, they appear somewhat differently to the programmer, and the AST representation for Tcl/Tk programs includes other types of nodes that are better candidates for elision (e.g., "group"). Since such syntactic categories are defined by the adaptation, their relevance to elision is not generally knowable by the generic tool code. Because StarTool used the text in star diagram nodes to determine elidability, the obvious stop-gap measure was to have the Tcl/Tk adaptation return star diagram labels for elidable nodes that looked like their C counterparts (an artifact from the original implementation).

The second problem concerned the kind of star diagram to be constructed. The original StarTool allows users to build star diagrams based on selected variables or all variables of a type (e.g., `int`). However, building type-based star diagrams makes little sense for a source language with dynamic typing. Other criteria for constructing star diagrams, such as all variables declared within a particular scope, prove to be more useful when analyzing Tcl/Tk programs.

Gnat Ada. To further explore language issues and assess whether the adaptation layer helps an adaptation programmer to easily leverage existing program representations, we chose to retarget the tool to Gnat, a public-domain Ada compiler that provides facilities for manipulating an AST representation of Ada95 source [3].

Retargeting to Ada exposed another language-dependent aspect of star diagram displays not accommodated by the adaptation interface. As described in Section 2, StarTool merges function and file nodes. Unlike C and Tcl/Tk, the modules of Ada programs are comprised of compilation units such as package and task bodies, rather than files, and subprogram constructs may be nested. Examination of star diagrams built from the Gnat AST revealed that merging these nested constructs would provide better information to the user.

The Gnat retarget also revealed shortcomings of the adaptation interface's AST traversal functions. For each kind of AST node, Gnat provides a unique set of functions to retrieve the child nodes. For example, an if-statement node's children are accessed via functions named `Condition`, `Else_Statements`, `Elseif_Parts`, and `Else_Statements`. Although implementing generic child and sibling accesses with these specific ones is straightforward, it is not efficient. The amount of tree traversal generated by translating from AST nodes to text positions and back slowed the tool unacceptably. This problem led us to enhance the adaptation implementation to cache generic child information along with the full source range for each node, computed during an initial walk of the AST. Caching consumes considerable space at runtime and adds over 700 lines of code to the adaptation module.

The Gnat retarget required approximately two weeks of work, the bulk of which was devoted to gaining a sufficient understanding of the Gnat AST. The implementation consists of approximately 2000 lines of code, twice the size of the adaptation for Ponder C. 700 lines of this excess can be attributed to making generic node references efficient, and a significant portion we attribute to the relative complexity of the Ada language definition.

5 DISCUSSION

Our retargets to C, Tcl/Tk, and Ada highlight a couple of successes and a couple of problems with our initial approach.

Adaptation effort. The size of the adaptations and the time it took to code them are reasonable. The adaptation to Ada is notable because it exploits an existing compiler intermediate representation to avoid the challenges of implementing an Ada AST ourselves. The interface contains 14 operations and the three adaptations required between 300 and 2000 lines of code each, none requiring more than two weeks work. Although the varying size and complexity of the adaptations is partially a reflection of the languages' relative complexity, two other factors played an important role.

First is the implementation of accurate AST–text mappings. Providing these mappings for Ponder C accounts for 250 lines of adaptation code. Although Gnat provides some positional information, producing accurate mappings from it required 350 lines of code. Accurate AST–text mapping could be an issue for many representations. Unfortunately, there is little chance of overcoming this problem with adapter interface design, since it is simply a missing feature that must be implemented.

Second, nearly a third of the Gnat retarget is due to an interface mismatch between the adaptation interface's requirement for first-child and next-sibling operations and Gnat's named-child functions. This mismatch is a likely problem in future retargets that could be addressed by interface redesign.

Adaptation suitability. Because the adaptation interface provides a label-generating function for star diagram nodes, star diagrams have the look and feel of the programming language. However, this success was only skin deep, especially when it came to diagram manipulation. The C, Tcl/Tk, and Ada programming languages vary with respect to what constructs should be elidable, mergeable, and also what kinds of star diagrams would be useful. Since the interface provides no way to express these variations, we had to force the language's features and some node labels into a C-like taxonomy.

In considering the code bloat in the Gnat adaptation caused by the mismatch between the our adaptation interface and the Gnat services, we realized that the `ast_sibling` and `ast_child` operations are present only to permit comparing each AST node to a candidate node (using adaptation interface function `similar`) to build the root set of the star diagram. Much of the code bloat could be eliminated, then, if the Gnat Ada adaptation had to provide only an AST node iterator, which would hide the distinction between accessing children by name and by order, giving adaptation layers more freedom in implementing AST traversals. This change eliminates over 500 lines of code from the Gnat adaptation without compromising performance. The effects are negligible on the Ponder C and Tcl/Tk adaptations, since they did not suffer interface mismatch.

For the language-feature mismatches, consider node elision in star diagrams. To elide nodes corresponding to a language-specific construct, the generic portion of the tool must permit the user to *select* the syntactic category of the construct for elision, and the tool must be able to *test* whether an AST node belongs to that category. Belonging to the class of elidable constructs can be viewed as a new, tool-specific AST node-type attribute, *elidable*, that the adaptation module implementation must synthesize from other AST node-type attributes.

Consequently, we added two new functions to the adaptation module interface for supporting this new attribute. The function `ast_elision_attributes` returns a set of AST node attributes—represented as a tab-delimited string—that make a node subject to elision. The attribute strings were given meaningful names, permitting the generic portion of the tool to display them directly in the star diagram elision panel. When the tool user selects one of these attributes, the generic tool code calls the other new adaptable interface function, `ast_has_attribute`, passing the elision attribute and an AST node. Nodes passing this test are elided from the display. Likewise, to support the tool-specific AST node attribute *mergeable*, we added the function `ast_merging_attributes` to the interface and extended the `ast_has_attribute` function to recognize the elements of the new set.

Implementing these functions was straightforward. For all adaptations the functions `ast_elision_attributes` and `ast_merging_attributes` each required one line of code, and `ast_has_attribute` required upto 25.

To permit the construction of unique kinds of star diagrams for each programming language, we replaced the fixed enumeration `SimilarityTypes` with adapter-defined AST similarity attributes by adding the function `ast_similarity_attributes` to the adaptation layer interface, which returns the attributes that may relate nodes. In turn we modified the generic portion of the tool to display these attributes directly to the user for selection, and the interface function `similar` was modified to accept these language-defined attributes.

Client-centric interface design
Unlike the initial set of language representation and semantics functions, which are low-level general-purpose AST-oriented operations, the new operations for supporting elision, merging, and comparison are both *tool-oriented* (they would have no useful function in a different tool) and serve to configure the tool to express the peculiarities of the programming language. This tool orientation is a departure from our initial expectations. Since star diagram node elisions, for example, are centered around compound statements, we might have chosen elision-helping operations such as `ast_isLoopStmt` and `ast_isCaseStmt`. However, this would not have provided for an open-ended, flexible, set of elision categories and would have constrained the tool to display fixed generic names for these (e.g., "Loops"), regardless of what special constructs a language like Tcl/Tk or Ada might contain.

This insight led us to reconsider our choice to provide a node iterator, which has a representation-independent, rather than a tool-centric, flavor. A basic iterator might be suboptimal for future adaptations because it cannot exploit a representation's precomputation of sets of similar nodes, for example in the form of definition-use chains or reference sets in the symbol table. Lacking a means to access such a direct representation, StarTool is forced to examine every node in the AST to gather the references itself.

A tool-centric approach suggests providing a function `next_similar` specifically for constructing star diagram root sets, thus absorbing the iteration and comparison duties into a single function and leaving the details of similar node collection to the implementation. Although the resulting interface restricts what StarTool can do to the representation, it is just this restriction that simplifies the adaptation code by permitting it to support less functionality.

Indeed, the generic StarTool component represents a single, fixed client implementation that we attach to multiple service implementations. This reverses the typical client–service layering relation, in which a single service implementation is designed to support multiple clients. By moving the adaptation interface away from the changing service side, we create a sufficient semantic gap between them to give the designers of adaptations the flexibility needed to take advantage of optimization opportunities in the services.

Consequently, we reformulated the adaptation layer's interface as a tool-centric (i.e., client-centric) *requires* interface

```
int al_elaborate(int &argc, char *argv[]);

char *al_elision_attributes();
char *al_merging_attributes();
char *al_similarity_attributes();

int su_has_attribute(SyntaxUnit item,
                     char *attribute);

/*
** Provides iteration of elements appropriately
** similar to #prototype# under/inside the
** #container#.
*/
SyntaxUnit first_similar_su(SyntaxUnit container,
                            SyntaxUnit prototype,
                            char *similarity);
SyntaxUnit next_similar_su();

/*
** Provides iteration of elements with #attribute#
** under/inside the #container#.
*/
SyntaxUnit first_su_with_attribute(
                    SyntaxUnit container,
                    char *attribute);
SyntaxUnit next_su_with_attribute();

/*
** Formerly the ast_parent operation.
*/
SyntaxUnit su_superunit(SyntaxUnit item);

/*
** Given a SyntaxUnit #item# and the #subunit#
** from which it was reached, returns a label
** indicative of #item#, possibly with an
** indication of which position #subunit#
** resides.  (Label modification was formerly
** performed in the tool, but was moved down
** along with other representation traversals.)
*/
char *su_label(SyntaxUnit item,
               SyntaxUnit subunit);

int su_skip_test(SyntaxUnit item);

struct FilePosition {
  int line, column;
};

char        *su_file(SyntaxUnit item);
FilePosition su_begins(SyntaxUnit item);
FilePosition su_ends(SyntaxUnit item);
SyntaxUnit   file_to_su(char *pathname);
char        *file_text(SyntaxUnit item);
char        *file_filters();
SyntaxUnit   file_range_to_su(
                   SyntaxUnit container,
                   FilePosition *range_begin,
                   FilePosition *range_end);
```

Figure 4: The reformulated StarTool-centric adaptation module interface, which contains 19 functions. The identifier sub-tag `al` stands for *adaptation layer*; the tag `su` stands for *syntax unit*. These were changed and generalized to reflect the greater independence from the representation.

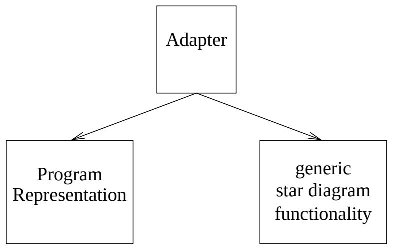


Figure 5: The *knows about* relation amongst the components of StarTool (shown as directed edges) denotes a mediator relation for the adapter, which is maintaining independence of the generic tool and program representations.

for StarTool that states exactly what it requires in terms of its own features (Figure 4). The interface has two parts: a *language interface* that serves to configure StarTool to accommodate the peculiarities of the programming language (the `al_` operations), and a *representation interface* that performs operations on the underlying representations (e.g., the `su_` operations), perhaps parameterized themselves by data retrieved earlier from the language interface.

The choice of a client-centric requires interface, although somewhat counter-intuitive, is actually an application of information-hiding modularity. The client, although not free-standing, should not attempt to exploit the implementation details of the services to which it will be connected. Should these details change (e.g., upon retargeting to an unanticipated language representation), either the client will have to be changed or the adaptation code will be made unnecessarily complex by attempting to satisfy the client's assumptions. In the language of Parnas, although the client *uses* the service and *depends upon* it for its correct functioning, it should not *know about* (make assumptions about) its design [12], in this case even its interface (Figure 5). It is the responsibility of the adaptation layer to translate between the independently developed service and client components by matching the requirements of their interfaces; the client imports functionality from the adaptation layer while dictating the interface. The adaptation layer hides the design decisions about how the translation is achieved. In this respect, the adaptation layer is a proper mediator module: it knows about both the provided interface of the service and the required interface of the client [14]. The fewer constraints that these two interfaces put on the environment, the easier it is to implement the mediator. In this respect, it is best for the client to require services in terms of its own features, not in terms of the features of a hypothetical service.

The conclusion discusses how to approach the design of client-centric interfaces from scratch and the situations in which it might be fruitfully applied.

Extensions for Reuse and Multi-Language Support
After completion of the tool-centric interface redesign, two

issues arose. First, each retarget contained some code similar to code in another retarget. This redundancy was due to the fact that although the new tool-centric interface minimizes assumptions about the underlying representations, it puts some extra, albeit small, responsibilities on the adaptations (e.g., aggregating similar nodes) because it is effectively operating at a higher level. Some of those responsibilities are handled identically by most retargets—for example retargets whose underlying representation provides generic child and sibling operations. Also, many adaptation interface operations admit trivial implementations for an initial retarget. For example, no retarget is required to support tool-centric attributes for elision. Second, we desired to support a multi-language tool, one that could handle a program written in a combination of, say, C and Tcl/Tk, which is a typical use of Tcl/Tk. However, the purely procedural adaptation interface does not readily permit multiple implementations of the same operation.

These issues were straightforwardly addressed with an object-oriented version of the tool-centric adaptation interface. We added a `StarAdapterClass` base class with default implementations of operations, both reducing redundancy and providing name space control. StarTool still imports the procedural interface, however, making no assumptions about an adaptation's ability to support object-orientation. The object-oriented implementation is merely a resource that a programmer can leverage to implement the procedural interface. Although this approach results in an extra layer of calls, inlining by the compiler is possible unless the adaptation subclass is selected dynamically at runtime.

`StarAdapterClass` provides default implementations for 14 of the interface operations, amounting to 160 lines of potentially reusable code. Most adaptations will reuse only a portion of this code, since many of the default implementations are essentially no-ops. However, these can help get an adaptation running quickly by permitting the adaptation programmer to focus initially on the few central representation operations.

Prototypes of multi-language tools for C and Tcl/Tk, as well as C, Tcl/Tk, and Ada, have proved straightforward to implement, requiring 450 and 550 lines of new adaptation code apiece, including the handling of cross-language variable linkages. The adaptation is implemented by writing glue code that combines existing adaptation classes into a single adaptation interface (Figure 6). The glue code has two primary responsibilities for any interface operation: (1) dividing a whole-program query from the generic tool into a query on each target implementation and (2) combining the results of those queries into a single return value.

A typical aspect of dividing a whole-program query is distinguishing the target representation of a particular `SyntaxUnit` that is passed as a parameter. This classi-

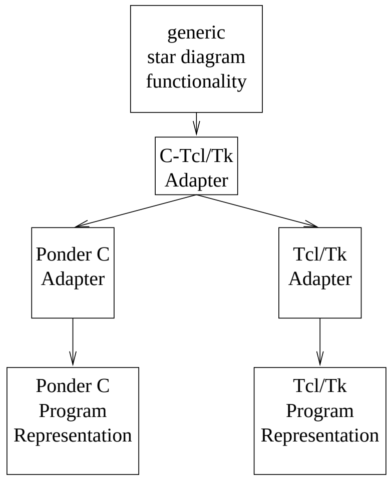

generic star diagram functionality
C-Tcl/Tk Adapter
Ponder C Adapter
Tcl/Tk Adapter
Ponder C Program Representation
Tcl/Tk Program Representation

Figure 6: Multi-language retarget of StarTool using adapter classes. Edges are function calls.

fication is currently implemented with STL `Map` [15] from `SyntaxUnit`'s to target implementations. Combining the results returned from a query across multiple targets often amounts to a union. Attribute operations are not so simple to handle, however, as they involve multi-language semantics. For example, should the programmer be allowed to separately control elision of C *if* statements and Tcl/Tk *if* statements, or should the two be treated as one construct? Although such questions are difficult to answer *in general*, an answer is simpler in the context of a single tool like StarTool: what choice better serves a user of StarTool? In defining elision categories, for example, the issues include the amount of control the user needs, user interface clutter (e.g., a potential explosion of elision categories), and the programmer's conception of similar constructs. Such issues could result in the creation of a new multi-language elision category that must be translated into the particular elision categories of the underlying adaptations.

6 RELATED APPROACHES

Common intermediate representation. The information required by a program analysis tool often closely resembles the information needed by a compiler. Retargetable compilers ease the process of adapting to both different languages and different machine architectures by defining a common intermediate program representation. Language-specific translators transform a source program into this common representation, then pass the result to language-independent components for additional translation. Although program analysis tools often use the same program representations as compilers, providing retargetability through the use of a common representation works poorly with these tools. Developing software to translate a new source language into a common representation can require months of effort because of the semantic detail required and coping with mismatches between the source language and the common representation. Compilers recover this investment by reusing the complex modules that perform optimization and translation to any number of target architectures. In contrast, development of the source translator can consume the bulk of the time required to write a new version of a program analysis tool. Also, programmers must be able to closely relate information produced by a software analysis tool to the source text in order to understand an existing software system. Common representations store information about the program in a language-neutral fashion, and the loss of language-specific detail from a common program representation limits the amount of information about the source text that a tool can display.

Genoa. Genoa, a software framework that allows rapid development of small, special-purpose program analysis tools, uses a variation of the common representation approach [2]. The process of instantiating Genoa for a new language involves writing a specification that allows translating queries of its language-independent program representation into queries of an existing compiler representation. GEN++, for example, is a Genoa instantiation that provides a query interface for C++ programs built on top of the cfront C++ translator. Although the structure of the Genoa AST is language-independent, the framework allows language-specific information to be embedded in the AST nodes, thus avoiding the problems with language-neutral representations.

Devanbu reports that Genoa instantiations generally require a 1000-2000 line specification and up to two months to develop. Much of the complexity of the instantiation process stems from the large amount of information that must be carried in the Genoa program representation to support the writing of a wide variety of analysis tools. To provide this information, the programmer responsible for adapting Genoa to a new language must learn the supporting compiler representation in detail and then write specifications that allow the translation of queries. Both of these steps can take considerable time for a complex program representation.

Our initial representation-centric approach to retargeting StarTool is analogous to the Genoa approach, except that we use a procedural approach, rather than a declarative approach, to prescribe the mapping between the underlying program representation and the tool's features. The procedural approach is likely less compact, but provides more flexibility. As we learned by retargeting to Gnat, the representation-centric approach is inappropriate when retargeting to support a single program analysis tool, as it entails more work than desired and can hurt performance.

7 CONCLUSION

Program analysis tools are more useful if they can process programs written in a variety of programming languages. We have experimentally developed a method for easing the adaptation of program analysis tools to new source languages, based on reusing existing program representations via an adaptation module that acts as a mediator between the representation and the program analysis tool.

The core of this method is the definition of a tool-centric interface that the generic program analysis client requires as a service from an adapter. A tool-centric interface has two logical parts: a language interface that customizes the tool's behavior with respect to the peculiarities of the programming language and a representation interface that queries and manipulates the actual program. This approach gives the implementor of an adaptation control over how unique language features are handled and flexibility to leverage peculiar features of the underlying representation to concisely and efficiently satisfy the tool's requirements. Reuse is enhanced by providing an adapter base class from which adaptations are subclassed. These classes can also be used to combine adaptations into a multi-language tool.

Two weeks of work and 300-1500 lines of adaptive code sufficed to retarget our adaptable version of StarTool to dissimilar representations of diverse source languages. Class-based versions of the adaptation interface permitted defining multi-language instantiations of StarTool that reuse existing retargets, requiring a week or so of additional work and about 500 lines of code.

Requires Interface Design Process

Compared to the state of the art in designing provides interfaces, the design of requires interfaces is relatively unexplored. Moreover, it is all too easy to fall into old ways and apply the techniques of normal interface design to requires interfaces. Based on our experience, the following process helps to avoid the missteps that we made, minimizing the requires interface's assumptions about the underlying representation service:

1. Identify the representation requirements of the analysis tool (e.g., tree traversal, mappings to program text).
2. Formulate these requirements in tool-centric terms.
3. Identify those tool features (e.g. elision) that have a behavior or presentation that varies across languages.
4. For each such feature, define a tool-centric attribute query function for the affected language elements (e.g., `al_elision_attributes`).
5. For each feature with varying *behavior*, define attribute functions to realize the necessary behavior (e.g., `su_has_attribute`). Different attribute categories might share functions, such as simple boolean attribute tests.
6. For each feature with varying *presentation*, define functions that realize the appropriate presentation. If the presentation is static, it can be achieved with values that also represent the attributes (e.g., strings), avoiding the inclusion of an additional function.

7. Finally, all representations of data should either be opaque references that are only processed through the adaptation interface, or least-common-denominator types like strings and integers.

Other Applications

We succeeded in applying this approach to the design of StarTool, a syntax-based analysis tool. However, the approach does not depend directly on program analysis, although the benefits are especially strong because of the plethora of programming languages in use today.

For example, we are currently applying this approach to the design of a user interface component for finite element analysis. The problem in this domain is that each finite element analysis system supports a specialized set of solvers. Over time, a user's changing modelling needs can require using different finite element systems. We are applying our client-centric design technique to this problem to reduce the cost of implementing multiple interfaces, save users from having to learn multiple interfaces, and ultimately integrate the tools into an integrated finite element system.

A determining factor in the cost effectiveness of the technique for a particular application is the amount of configurability that must be supported by the client to accommodate unique properties of services. Although the StarTool user interface is effectively parameterized with respect to language specifics via tool-centric attributes, it does not support radical reconfigurations of the interface. Elbereth, a star-diagram based tool for Java has a somewhat different organization [9], which StarTool's requires interface cannot currently support. Of course, limitations are a necessary property of interfaces; the line must be drawn on functionality somewhere.

Acknowledgments. We thank Kevin Sullivan for his insights on how our approach relates to mediator design. We also thank all the people who have contributed to the StarTool implementation, including Darren Atkinson, Morison Chen, Jenny Cabaniss, Lee Carver, Walter Korman, David Morgenthaler, and Van Nguyen.

REFERENCES

[1] R. W. Bowdidge and W. G. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Transactions on Software Engineering and Methodology*, 7(2):109–157, April 1998.

[2] P. Devanbu. GENOA – a customizable, language- and front-end independent code analyzer. In *Proceedings of the 14th International Conference on Software Engineering*, pages 307–317, May 1992.

[3] R. B. K. Dewar. The GNAT model of compilation. In *Proceedings of Tri-Ada '94*, pages 58–70, November 1994.

[4] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[5] W. G. Griswold and D. C. Atkinson. Managing the design trade-offs for a program understanding and transformation tool. *Journal of Systems and Software*, 30(1–2):99–116, July–August 1995.

[6] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceeedings of the IEEE 1996 Workshop on Program Comprehension*, pages 144–153, March 1996.

[7] W. G. Griswold, M. I. Chen, R. W. Bowdidge, J. L. Cabaniss, V. B. Nguyen, and J. D. Morgenthaler. Tool support for planning the restructuring of data abstractions in large systems. *IEEE Transactions on Software Engineering*, 24(7):534–558, July 1998.

[8] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.

[9] W. F. Korman. Elbereth: Tool support for refactoring java programs. Masters Thesis, University of California, San Diego, Department of Computer Science and Engineering, June 1998. Technical Report CS98-590.

[10] H. A. Muller, S. R. Tilley, M. A. Orgun, B. D. Corrie, and N. H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *Proceedings of the SIGSOFT '92 Fifth Symposium on Software Development Environments*, pages 88–98, December 1992.

[11] J. K. Ousterhout, editor. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.

[12] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, March 1979.

[13] K. J. Sullivan and D. Notkin. Reconciling environment integration and component independence. In *Proceedings of the SIGSOFT '90 Fourth Symposium on Software Development Environments*, pages 22–33, December 1990.

[14] K. J. Sullivan and D. Notkin. Reconciling environment integration and component independence. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[15] M. J. Vilot. An introduction to the Standard Template Library. *C++ Report*, 6(8):22–29, 35, October 1994.