

Designing for Retargetability of a Program Analysis Tool by Reusing Intermediate Representations*

James Hayes, William G. Griswold, Stuart Moskovic
Department of Computer Science and Engineering, 0114
University of California, San Diego
La Jolla, CA 92093-0114
{jhayes, wgg, stuart}@cs.ucsd.edu

Abstract

An interactive program analysis tool is most useful if it can be easily adapted to different source languages. However, these tools are often tailored to one particular representation of programs, making adaptation to a new language costly. An inexpensive way of achieving adaptability is to introduce an intermediate abstraction—an adaptation layer—between an existing language representation and the program analysis tool that translates the tool’s queries into queries on the particular representation.

We experimented with this approach in making our StarTool program analysis tool retargetable. The approach proved successful, resulting in low-cost retargets for C, Tcl/Tk, and Ada. However, some unanticipated adjustments to the approach were required, providing insights for making a client retargetable. First, the adaptation layer exports two interfaces, a representation interface that supports queries on the represented program and a language interface that the client queries to configure itself to behave appropriately for the given language. Second, retargeting is eased if the program analysis tool is designed to import a tool-centric (i.e., client-centric) interface rather than a general-purpose, language-neutral language representation service interface. Straightforward object-oriented extensions enhance reuse and facilitate the development of multi-language tools.

1 Introduction

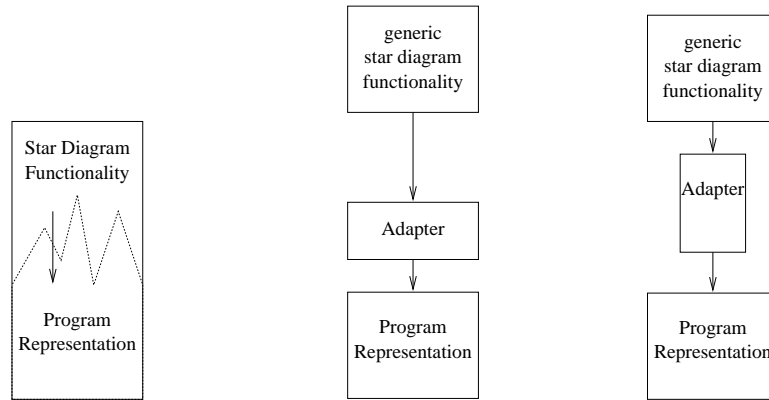
By summarizing information that is collected from a software system as a whole, a program analysis tool can reduce the time required by a programmer to understand a system well enough to begin making changes. For example, a tool like RIGI can summarize the architectural elements of a system [Muller et al. 92]; our StarTool provides hierarchical, cross-cutting views of all the uses of a data structure or other design decision [Griswold et al. 98].

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 adaptation layer—between an existing language representation and the program analysis tool that translates the tool’s queries into queries on the particular representation [Gamma et al. 95, pp. 139–150]. Genoa [Devanbu 92] 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. Only a small number of operations should be required because of StarTool’s limited needs. We restructured StarTool to remove representation-specific references in the code and have it import such an interface in their place, and developed retargets to the Ponder C program representation [Griswold & Atkinson 95], a similar representation for Tcl/Tk, and the Gnat Ada program representation [Dewar 94] (Figure 1b).

*This work supported in part by UC MICRO grant 97-061 joint with Raytheon Systems Company.



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

Figure 1: Evolution of StarTool’s design for retargetability. Directed edges denote the uses relation, as realized by function call. Positioning of the adapter box reflects the closeness of its interfaces to the components it connects. Its shape reflects the scope of its responsibilities.

Although these representations are indeed tree-based, the export of a low-level and supposedly 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 that we had treated 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 must *import* functionality in order to function at all. Hence, the effort in supporting a low-level interface was misplaced: the client did not need such an interface and so the adaptation layer did not need to support it. An interface that was both representation-oriented and language-neutral also prohibited conveying language-specific to the tool and ultimately the tool user. The implication, then, was that we should be thinking in terms of designing an appropriate *requires* interface for the client, which states only what the tool requires in terms of the tool’s features (Figure 1c).

This choice explicitly requires the adaptation code to understand the relationship between the language representation and the tool’s features. Moreover, the resulting interface is “high-level” in the sense that it is expressed in terms of the tool’s specific needs. 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 by permitting 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, the adaptation layers function as glue-hiding mediators that serve to combine independently developed components [Sullivan & Notkin 90, Sullivan & Notkin 92].

Section 2 describes StarTool and its representation requirements. Section 3 describes our initial retargetable interface and the Ponder C retarget, followed by the retargets for Tcl/Tk and Gnat Ada as well as the problems encountered. In section 4 we discuss our insights and the resulting revised adaptation interface. Section 5 discusses improving reuse, including support for multi-language retargets. We close with consideration of related retargeting approaches, a review of our method, and a discussion of open problems.

2 StarTool and Its Representation Requirements

The StarTool program analysis tool assists programmers in planning restructuring projects on large software systems [Griswold et al. 98]. In particular, it helps in making 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

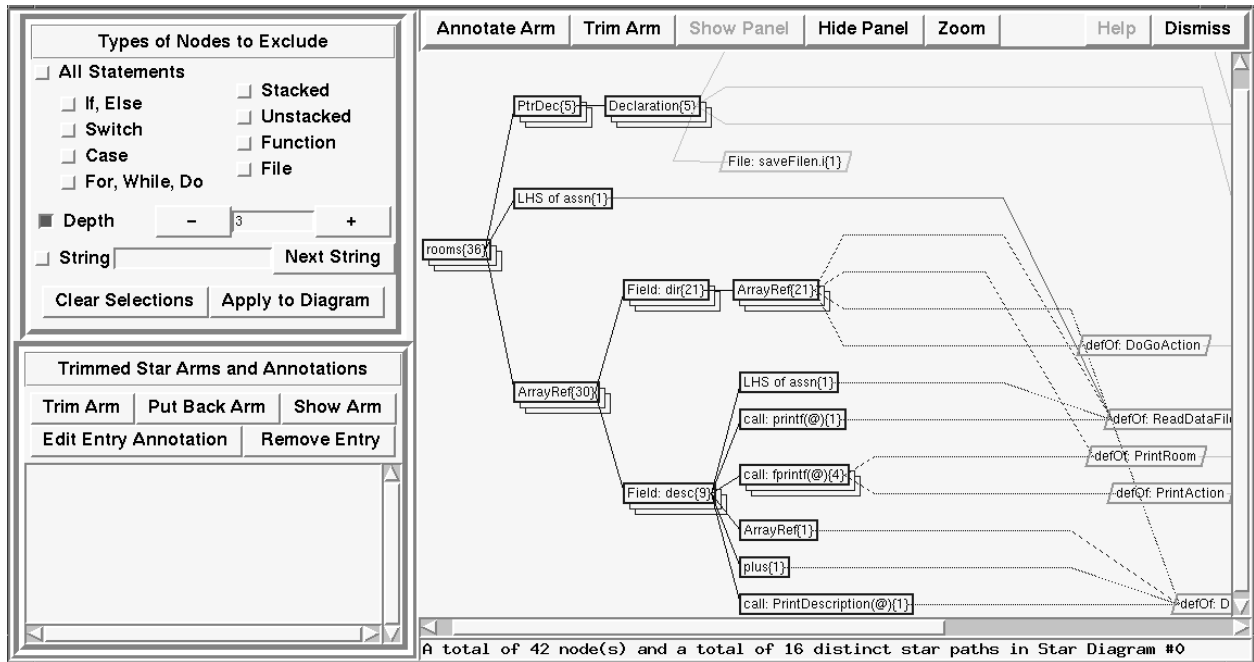


Figure 2: A star diagram window for the variable rooms.

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 are nodes representing 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 inspect the program text associated with a node. Restructuring planning 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.

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 nodes 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) [Bowdidge & Griswold 98]. 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, same variable or same type. The second stage computes the descendants of the star diagram root, which involves 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 all the leaves of the star diagram are file nodes. The third stage computes a graphical layout of the star diagram, including the merging of the function and file nodes.

3 Initial Restructuring and Adaptations for C, Tcl/Tk, and Ada

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 AST's [Griswold & Notkin 93, Griswold & Atkinson 95], 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, providing low-level functionality would not unnecessarily burden adaptations with high-level responsibilities, and lack of language bias would not unnecessarily complicate 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.

We proceeded in three stages. First, we restructured the existing tool to isolate representation and language semantics issues in an adaptation layer. Second, we challenged this design by retargeting to a rather different language. Finally, we challenged the design further with a third retarget to a rather different representation and language.

3.1 Tool Restructuring and Ponder C Retarget

The immediate goal of the first phase was to retarget the tool from a tool-specific, C program AST representation to C program AST's generated by the Ponder language toolkit. Ponder combines a yacc-like grammar specification tool with data structure libraries and facilities for manipulating generic program AST's and symbol tables [Griswold & Atkinson 95]. The C instantiation of Ponder existed before we began restructuring [Griswold et al. 96].

Isolating program representation details with an intervening adaptation module required restructuring [Griswold & Notkin 93] 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: 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 necessary semantic queries on the representation to support the construction of meaningful star diagrams, a small set of generic query functions are provided. 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 an adaptation to use a particular character might complicate label production for some adaptation.

Since the programmer works with program text as well as star diagrams, functions are provided 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 what text positions 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. The tool is required to call this prior to its own initialization.

The resulting restructuring into generic StarTool and adaptation components resulted in 1000 lines of non-blank, non-comment adaptation code for the 14 functions. A quarter of this code implements accurate and efficient AST-text mappings, since the Ponder C implementation provides only file and line number mappings for AST nodes.

3.2 Tcl/Tk Retarget: Assessing Language Retargetability

We next moved to the issue of evaluating the initial adaptation layer interface for retargetability with respect to language issues, which were difficult to thoroughly understand in the context of a single language. We decided to retarget the tool to support programs written in the rather different Tcl/Tk [Ousterhout 94]. The representation for Tcl/Tk programs itself was designed to minimize representation adaptation issues, permitting a focus on language issues.

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

```

/*
** 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 will replace 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 and file mapping.
*/
char *ast_file_text(AstNode item);
char *ast_file(AstNode item);
AstNode ast_file_AstNode(char *pathname);
FilePosition ast_begins(AstNode item);
FilePosition ast_ends(AstNode item);
AstNode ast_find_AstNode(AstNode start_node, FilePosition start, FilePosition end);

```

Figure 3: The adaptation module interface after the retarget to the Ponder C implementation.

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”). Although such syntactic categories are defined by the adaptation, their relevance to elision is not generally knowable by the generic tool code. To elide nodes corresponding to a unique construct, the generic portion of the tool must permit the user to select the syntactic category 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 attribute, *elidable*, that the adaptation module implementation must synthesize from other AST node attributes.

We added two new functions to the adaptation module interface to support specifying and testing this new attribute. The `ast_elision_attributes` function returns a set of AST node attributes—represented as a tab-delimited string—that make a node subject to elision. If the attribute strings are given meaningful names, the generic portion of the tool can display them, without interpretation, 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. Implementing these functions was trivial. In both the C and Tcl/Tk versions of the tool `ast_elision_attributes` required one line of code and `ast_has_attribute` required less than 10.

The original StarTool allows users to build star diagrams based on selected variables or all variables of a type. However, building type-based star diagrams makes little sense for a source language with dynamic typing. On the other hand, other criteria for constructing star diagrams, such as all variables declared within a particular scope, prove to be useful when analyzing Tcl/Tk programs. This led us to conclude that the criteria for including an AST node in a star diagram root set should be considered language-dependent. Consequently, we replaced our fixed `SimilarityTypes` enumeration with tool-defined AST similarity attributes, adding 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.

With these changes in place, we completed the Tcl/Tk retarget in about a week. The Tcl/Tk adaptation module contains 300 lines of code. 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.

Unlike the initial set of language representation and semantics functions, which are low-level general-purpose AST-oriented operations, the two new operations are tool-oriented and likely would have no useful function for a different tool. 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 language might contain. We revisit this issue in Section 4.

3.3 Gnat Ada Retarget: Assessing Representation Retargetability

We next turned to assessing whether the adaptation layer provides an interface that helps adapters to easily leverage existing program representations. We chose to retarget the tool to the public-domain Gnu Ada compiler, Gnat, which provides facilities for manipulating an AST representation of Ada95 source [Dewar 94].

Early on, retargeting to Ada exposed another language-dependent aspect of star diagram displays. 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. This led us to conclude that the adaptation module needed to support a new 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.

The Gnat retarget also revealed shortcomings of the AST traversal functions of the adaptation module interface. 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 also adds over 700 lines of code to the adaptation module.

This misfit between AST models reveals that the adaptation module’s representation operations provide no way to exploit opportunities for optimization specific to a program representation. For example, StarTool uses the child and sibling traversal functions to gather similar nodes (such as all references to an identifier) into a star diagram root set. This information may be available directly within the program representation, for example, in the form of definition-use chains or reference sets in the symbol table. Lacking a means to access this direct representation, the tool will visit every node in the AST in order to gather the references itself. We address this problem in the next section.

The Gnat retarget required approximately two weeks of work, the bulk of which was devoted to gaining 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, the rest we attribute to the relative complexity of the Ada language definition.

4 Discussion

Our retargets to C, Tcl/Tk, and Ada highlight several successes and a couple of problems with our initial approach. First, the size of adaptations and the time it takes to code them is reasonable. Our adaptation to Ada is notable because it exploited an existing compiler intermediate representation to avoid the challenges of implementing an Ada AST ourselves. The interface contains just 17 operations (14 original plus 3 added attribute-testing functions), 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.

One contributor 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.

A third source of variation in the Gnat retarget was attempting to avoid the runtime cost of expressing generic child and sibling references in terms of the node-type-specific functions that Gnat provides. The cause is that the adaptation layer interface provides a poor mapping between the facilities in Gnat and the needs of StarTool: generic node references are hard to implement efficiently with Gnat.

In considering this mismatch, we realized that much of the problem could have been avoided if the Gnat Ada adaptation code only had to provide an AST node iterator, rather than the lower-level `ast_sibling` and `ast_child` operations. In short, the lower-level operations provide flexibility to StarTool—it can use them any way it wants—but it overconstrains the adaptation layer’s choices of how to iterate over the nodes. Consequently, we considered replacing the lower level operations with a node iterator. However, we realized that a simple iterator would not be able to exploit a representation’s precomputation of sets of similar nodes.

The solution to this problem lies in the approach taken in improving StarTool’s independence from the source language by providing information that is specific to a particular StarTool feature, rather than generalizing language features. Likewise, a tool-centered approach to representation independence provides, for example, `anext_similar` function specifically for constructing star diagram root sets, replacing the lower-level `ast_child`, `ast_sibling`, and `similar` in the adaptation module interface, thus leaving the details of similar node collection to the implementation. Such a function supports the optimization based on definition-use chains mentioned in Section 3. Although the resulting interface restricts what StarTool can do to the representation, it is just this restriction that simplifies the adaptation code—which is rewritten for each retarget—by permitting it to support less functionality.

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

Consequently, we reformulated the adaptation layer’s interface as a “high-level”, tool-centric (i.e., client-centric) *requires* interface for StarTool that states exactly what it requires in terms of its own features (Figure 4). The redesign eliminates over 500 lines of code from the Gnat adaptation without compromising performance. The effects on the Ponder C and Tcl/Tk adaptations was negligible, since their initial implementations did not suffer interface mismatch.

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

```

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

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

/*
** 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 18 operations. 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.

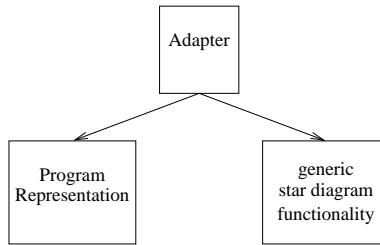


Figure 5: The knows-about relation amongst the components of StarTool (shown as directed edges) denotes a mediator relation for the adapter, which is maintaining independence of the generic tool and program representations.

unanticipated language representation), either the client will have to be changed or the adaptation code will be made unnecessarily complex by attempting 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 [Parnas 79], 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 and to match the requirements of their interfaces, even for the client that is importing functionality from the adaptation layer. It hides the design decisions about how this translation is achieved. In this respect, the adaptation layer is a proper mediator module: it knows about both the exported interface of the service and the imported interface of the client [Sullivan & Notkin 92]. 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 features of hypothetical services.

5 Extensions

After completion of the tool-centric interface redesign, two issues arose. First, we observed that each retarget contained some code similar to code in another retarget. This redundancy was due to the fact that although the new high-level interface minimizes assumptions about the underlying representations, it puts some extra, albeit small, responsibilities on the adaptations. 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 specific tool-centric attributes for elision, node comparison, etc. An empty or very small set suffices, at least during initial development of the retarget. Second, we desired to support multi-language tools, for example, a tool that could handle a program written in a combination of C and Tcl/Tk, which is a typical application of Tcl/Tk. However, the adaptation interface, being purely procedural, does not readily permit multiple implementations of the same operation.

We decided to address these two problems with an object-oriented version of the tool-centric adaptation interface. An object-oriented implementation with a `StarAdapterClass` base class provides default implementations of operations to reduce redundancy and coding effort, as well as 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. This choice does result in an extra layer of calls, but the class-level ones can be inlined by the compiler since the member functions (i.e., methods) called can be determined at compile-time by specifying the particular class (e.g., language) implementation being referenced (e.g., `GnatStarAdapter::next_similar_su`).

`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 access and mapping operations.

Prototypes of multi-language tools for C and Tcl, as well as C, Tcl, and Ada, have proved straightforward to implement, requiring 300 and 400 lines of new adaptation code apiece. 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.

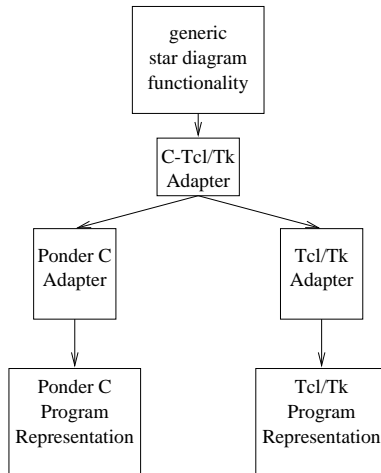


Figure 6: Multi-language retarget of StarTool using adapter classes. Edges are function calls.

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 classification is currently implemented with `STL Map` [Vilot 94] 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 more directly involve multi-language semantics. For example, should the programmer be able to separately control elision of C if statements and Tcl/Tk if statements, or should the two constructs be treated identically? Although such questions are difficult to answer *in general*, the solution is simpler in the context of a single tool like StarTool: what choice better serves a user of StarTool? Particular issues to consider for elision categories are how much control the user needs to be given, management of user interface clutter (e.g., a potential explosion of elision categories), and how the programmer perceives the similar constructs. Such choices could result in the creation of an entirely new multi-language elision category that then 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 source programs 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 [Devanbu 92]. 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 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 that the generic StarTool client imports a tool-centric interface, rather than a representation-centric interface, which gives an adaptation module implementor more control over how unique language features are handled and more flexibility to take advantage of particular features of the underlying representation that can efficiently satisfy tool requests. Reuse is enhanced by providing an adapter base class and encapsulating adaptations in subclasses. These classes can also be used to combine the adaptations into multi-language implementations.

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. Using class-based versions of the adaptation interface, multi-language instantiations of StarTool that reuse existing retargets is straightforward, requiring only a couple of days' additional work and 300-400 lines of code.

Our process for making a program analysis tool retargetable can be summarized as follows:

1. Identify the representation requirements of the analysis tool (e.g., tree iteration, mappings to program text).
2. Formulate these requirements in tool-centric terms.
3. For each tool feature, identify how the feature's behavior and presentation may vary across languages.
4. For each such feature, identify relevant tool-centric attributes for language elements affected by this feature.
5. For each feature with different behavior, define attribute functions to realize the necessary behavior.
6. For those features with varying presentation, define functions that realize the appropriate presentation. If the presentation is static, these can be represented as strings that also represent the attributes to avoid the inclusion of additional functions.

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

This work raises a few questions. How general is this technique: Does it apply to software other than program analysis tools? What if it was desired to attach StarTool to a more complete programming environment: How could StarTool surrender its text widgets and other "generic" windowing functionality to the environment? Lastly, although the StarTool user interface is effectively parameterized with respect to language specifics via tool-centric attributes, it is possible that some language could be sufficiently different that a different user interface *organization* would be desirable. For example, Elbereth, a star-diagram based tool for Java has a somewhat different organization [Korman 98]. Can such organizational issues be managed in a tool-centric fashion as well?

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

- [Bowdidge & Griswold 98] 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), April 1998.
- [Devanbu 92] 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.
- [Dewar 94] R. B. K. Dewar. The GNAT model of compilation. In *Proceedings of Tri-Ada '94*, pages 58–70, November 1994.
- [Gamma et al. 95] E. Gamma, R. Helm, J. Vlissides, and R. E. Johnson. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [Griswold & Atkinson 95] 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.
- [Griswold & Notkin 93] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology*, 2(3):228–269, July 1993.
- [Griswold et al. 96] W. G. Griswold, D. C. Atkinson, and C. McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proceedings of the IEEE 1996 Workshop on Program Comprehension*, pages 144–153, March 1996.
- [Griswold et al. 98] 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.
- [Korman 98] 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.
- [Muller et al. 92] 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*, December 1992.
- [Ousterhout 94] J. K. Ousterhout, editor. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading, MA, 1994.
- [Parnas 79] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, 5(2):128–138, March 1979.
- [Sullivan & Notkin 90] K. J. Sullivan and D. Notkin. Reconciling environment integration and component independence. In *Proceedings of the SIGSOFT '90 Fourth Symposium on Software Development Environments*, December 1990.
- [Sullivan & Notkin 92] 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.
- [Vilot 94] M. J. Vilot. An introduction to the Standard Template Library. *C++ Report*, 6(8):22–29, 35, October 1994.