A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates

Kathleen A. Lindlan, Janice Cuny, Allen D. Malony, Sameer Shende
Department of Computer and Information Science
University of Oregon, Eugene, OR 97403
{klindlan, cuny, malony, sameer}@cs.uoregon.edu

Bernd Mohr
Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich GmbH, D-52425 Jülich, Germany
B.Mohr@fz-juelich.de

Reid Rivenburgh, Computer Research and Applications Group
Craig Rasmussen, Advanced Computing Laboratory
Los Alamos National Laboratory, Los Alamos, NM 87545
{reid, rasmussn}@lanl.gov

Abstract

The developers of high-performance scientific applications often work in complex computing environments that place heavy demands on program analysis tools. The developers need tools that interoperate, are portable across machine architectures, and provide source-level feedback. In this paper, we describe a tool framework, the Program Database Toolkit (PDT), that supports the development of program analysis tools meeting these requirements. PDT uses compile-time information to create a complete database of high-level program information that is structured for well-defined and uniform access by tools and applications. PDT's current applications make heavy use of advanced features of C++, in particular, templates. We describe the toolkit, focussing on its most important contribution -- its handling of templates -- as well as its use in existing applications.

1 Introduction

Increasingly, high-performance scientific applications are being developed for complex computing environments where parallel and distributed code executes across heterogeneous platforms. The programs may use multiple languages, frameworks, libraries, and run-time systems, and they often depend on state-of-the-art hardware and software configurations that are constantly changing. The Advanced Computing Laboratory (ACL) [2] at Los Alamos National Laboratory (LANL) is typical of such environments. Its researchers are using advanced software technology (including object-oriented frameworks, scalable run-time systems, and component architectures) to develop a robust programming environment for computationally-intensive scientific simulations.

Software developers at ACL need program analysis tools that provide, at a minimum, the static and dynamic analysis capabilities (compilation facilities, debuggers, profilers, *etc.*) found in traditional sequential environments. They must also function across a diverse set of application programs, and integrate easily with new tools as they are developed. Portability across a variety of platforms is important. Further, program analysis should be performed at a level of abstraction that matches the programming models used and gives feedback in terms of source code constructs.

In this paper, we describe a tool infrastructure, the Program Database Toolkit (PDT), that supports the development of program analysis tools satisfying these requirements. PDT uses compile-time information to create a complete database of high-level program information that is structured for well-defined and uniform access by tools and applications. The high-level program information enables the construction of tools that operate at an appropriate level of abstraction. The use of compile-time, machine-independent intermediate representations enables the construction of tools that are portable. The uniform access mechanisms enable the construction of tools that can interoperate with other tools and applications.

PDT is targeted to the current-generation pro-

0-7803-9802-5/2000/$10.00 (c) 2000 IEEE.

```
template <class Object>
class Stack {
public:
    explicit Stack( int capacity = 10 );

    bool isEmpty( ) const;
    bool isFull( ) const;
    const Object & top( ) const;

    void makeEmpty( );
    void pop( );
    void push( const Object & x );
    Object topAndPop( );

private:
    vector<Object> theArray;
    int topOfStack;
};

template <class Object>
bool Stack<Object>::isFull( ) const {
    return topOfStack == theArray.size( ) - 1;
}
```

```
template <class Object>
void Stack<Object>::push( const Object & x ) {
    if( isFull( ) )
        throw Overflow( );
    theArray[ ++topOfStack ] = x;
}

template <class Object>
Object Stack<Object>::topAndPop( ) {
    if( isEmpty( ) )
        throw Underflow( );
    return theArray[ topOfStack-- ];
}

int main( ) {
    Stack<int> s;

    for( int i = 0; i < 10; i++ )
        s.push( i );
    while( !s.isEmpty( ) )
        cout << s.topAndPop( ) << endl;
    return 0;
}
```

Figure 1. C++ template definitions for an array-based `Stack` class and some member functions [19].

gramming languages commonly used in scientific computing: C++, Fortran 90, and Java. The use of C++ for the development of ACL software frameworks presented a critical test case for PDT. Thus, the first version of PDT includes analysis of the advanced object-oriented constructs of C++, such as multiple inheritance, namespaces, and template instantiations and specializations. Templates posed challenging analysis problems, and their treatment is a focus of this paper.

In Section 2, we discuss the difficulties in supporting C++ template analysis. In Section 3, we describe the toolkit and its handling of templates. Existing applications of PDT are presented in Section 4. In Section 5, we discuss research related to PDT and, finally, in Section 6, we assess the results of our work and outline future directions.

2 Challenges of Template Analysis

Templates are important constructs for object-oriented software because they permit compile-time polymorphism and generic programming. Consider the templated code (shown in Figure 1) that defines a `Stack` class to be created with any basic or user-defined type (*e.g.*, `double` or `Integer`) by substitution of the `Object` parameter. Each instantiation of a template yields object code that handles one particular type. Creation of template entities requires handling that works not only for single instantiations of arbitrarily complex types, but also for multiple instantiations and specializations.

Knowledge of template class definitions is useful for certain static source analyses, but other analyses will need to know how templates are instantiated. Unless the front end or compiling system makes this information available, it will be difficult for tools to determine the instantiation results from object code. Unfortunately, because templates are one of the more recent and complex C++ features, their support by compilers and analysis toolkits is frequently missing or inadequate [9]. In addition, most compilers do not provide easy access to intermediate program information.

The Edison Design Group (EDG) [1][12] C++ Front End is production-quality software that is nearly up-to-date with the C++ standard [6][18] in its support of templates, namespaces, exception handling, and pragmas. The EDG Front End outputs a high-level intermediate language (IL) that preserves the information available in source code, including original names and locations. By default, the EDG Front End instantiates templates with an automatic scheme. Compiling source files generates object files and template information files indicating potential instantiations. At link time, when the prelinker encounters references to

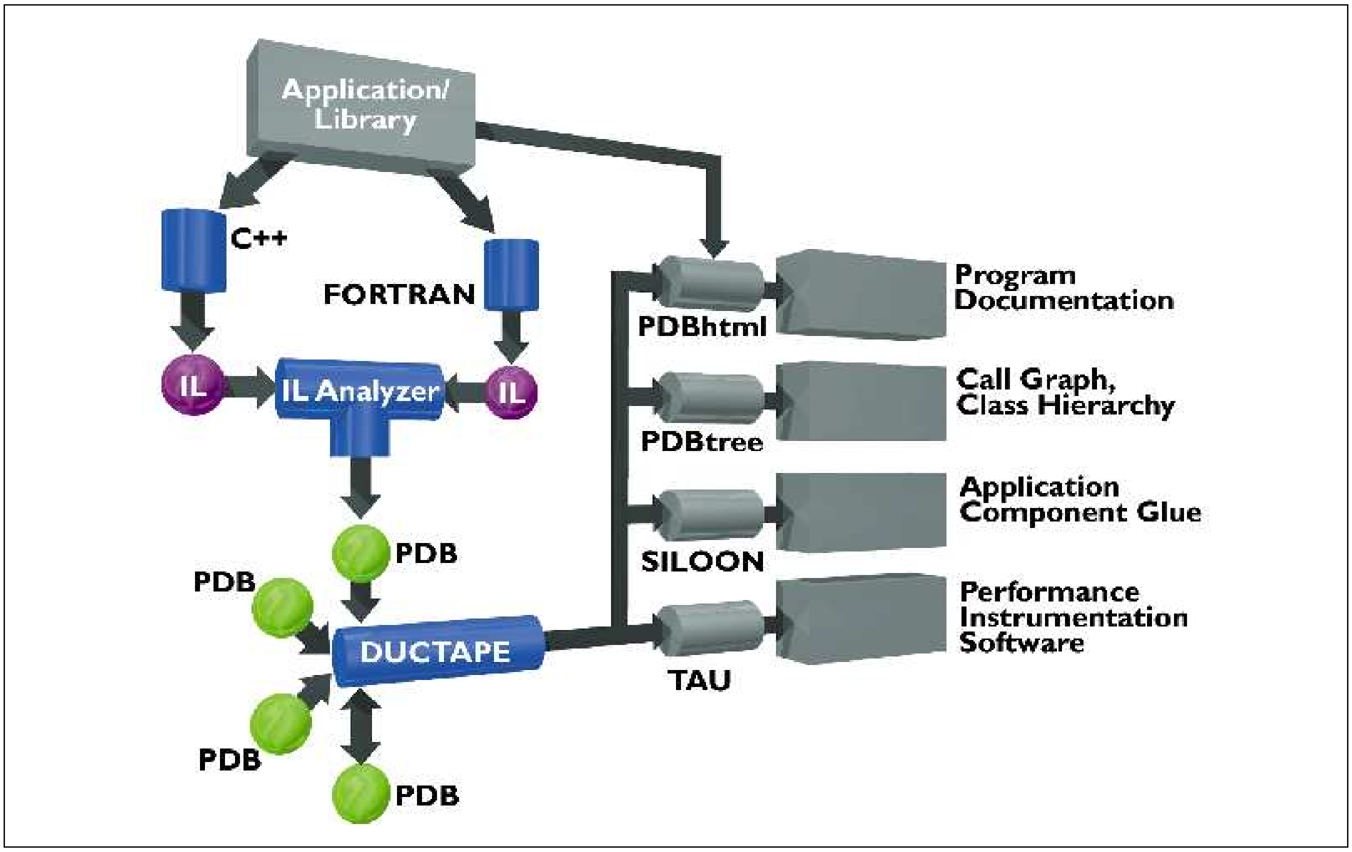

Figure 2. Source code is parsed by compiler front ends. PDT's IL Analyzers process the resulting intermediate language trees. PDT's DUCTAPE library makes the contents of PDB files available to applications. Existing applications are shown in the right half of the figure.

undefined template entities in object files, instantiations are assigned to instantiation request files. The source files needed for instantiation are then re-compiled. These steps continue until all templates are instantiated. Unfortunately, this process does not record and instantiate templates in the IL, where information is accessible by an analysis tool. The EDG Front End does, however, provide additional mechanisms for more precise control of the instantiation process. The "used" instantiation mode, enabled via a command-line option, provides the alternative needed by PDT. All template entities used in the compilation are instantiated and represented in the IL; unused member functions and static data members are not instantiated unnecessarily, minimizing compilation time and the size of the IL. Given the access to needed information for even the advanced language features, the EDG IL provides a useful starting point for PDT.

3 Program Database Toolkit

PDT provides applications easy access to high-level language constructs used in source code. It accomplishes this by filtering and reorganizing the information represented in the intermediate language trees produced during compilation by the EDG C++ Front End. Figure 2 [4] shows the PDT framework and its primary components. The first component, the IL Analyzer [3][14], walks the IL tree, extracting the high-level interface and outputting item descriptions to a program database. These descriptions characterize the program's functions and classes, including template instantiations, as well as templates, other types, namespaces, macros, and source files. The second component, the DUCTAPE (C++ program Database Utilities and Conversion Tools APplication Environment) [3] library, provides an API to the database.

PDT is designed to support different programming languages by utilizing multiple language-specific front ends and IL Analyzers. The program database

Item Type	Attributes of Item	Prefix
all ITEMs	source position	
HEADER	`<PDB 1.0>`	
SOURCE FILES	files included by source file	`so`
ROUTINES	template from which instantiated, parent class or namespace, access mode, signature, functions called, characteristics specifying linkage, storage class, virtuality, *etc*.	`ro`
CLASSES	template from which instantiated, parent class or namespace, access mode, direct base classes, friend classes and functions, characteristics, member functions, information on other members, including access, kind, and type	`cl`
TYPES	parent class or namespace, access mode, various characteristics, depending on type: *e.g.,* for function types, return type, parameter types, presence of ellipsis, and exception class IDs	`ty`
TEMPLATES	parent class or namespace, access mode, kind, text of template	`te`
NAMESPACES	members of namespace or alias	`na`
MACROS	kind, text of macro	`ma`

Table 1: Program Database (PDB) Item Types, Attributes, and Prefixes

format is intended to support common structures across languages as well as language-specific constructs. The PDT architecture as it exists for C++ is described further in this section.

3.1 IL Analyzer

The IL Analyzer processes an IL file to produce a human-readable "program database" (PDB) containing information on high-level source constructs (including source code locations). To do this, it traverses the IL tree, reporting information on designated, high-level constructs as they are encountered. Separate traversals for source files, routines, types, classes, namespaces, templates, and macros allow selection of the constructs to be reported, and prepending of distinguishing prefixes for common item attributes.

In processing a node, all related nodes are processed as well, so that a construct's attributes are completely summarized in one entry. For example, in order to report the functions a routine calls with other information on the routine, the file's IL tree is traversed until a routine declaration is encountered, at which time traversal switches to that routine's tree. Considerable processing is required for some constructs. Calls for constructors and destructors are not treated as standard routine calls by the Front End, since these routines are associated with objects having "lifetimes." PDT must process all contexts in which the lifetimes are handled in order to determine the calling locations. In some cases, extra processing is needed because the IL was designed as input to a compiler back end, not the IL Analyzer. Locations for some constructs are maintained in supplemental data structures that must be scanned, since they are not directly connected to the IL constructs being processed.

Templates must be handled carefully. An EDG instantiation mode that forces instantiation at compile time is used so that instantiation information can be available to PDT. IL subtrees are incorporated in the IL tree for each instantiated class and routine, which are then accessible to the IL Analyzer. The IL subtrees indicate that an entity has been instantiated, not the template from which it is derived. To compensate for this, the IL Analyzer creates a list of templates in advance, and then scans it to determine the template corresponding to an instantiation's locations. Because the location of a specialization is not within the associated template's definition, it is currently not possible to determine the originating template for a specialization. To remedy this, template IDs would have to be included in the IL constructs for instantiations and specializations, which would require modification of the EDG Front End.

3.2 Program Database

The IL Analyzer outputs item descriptions for relevant programming language entities: source files, routines, classes and other types, templates, namespaces, and macros. Each description identifies an item and lists its features. The identifier prefix indicates the type of language construct: *e.g.*, "`ro#7`"

```
<PDB 1.0>                     (1)

so#66 StackAr.h               (2)
sinc so#71
sinc so#72
sinc so#73

so#71 /pdt/include/kai/vector.h
                              (3)
so#72 dsexceptions.h          (4)

so#73 StackAr.cpp             (5)

so#75 TestStackAr.cpp         (6)
sinc so#66

te#559 Stack                  (7)
tloc so#66 23 15
tkind class
ttext template <class Object>
    class Stack {...};
tpos so#66 22 9 NULL 0 0
    so#66 23 9 so#66 40 9

te#566 push                   (8)
tloc so#73 72 14
tkind memfunc
ttext template <class Object>
    void Stack <Object>::
    push(const Object & x) {...}
tpos so#73 71 9 NULL 0 0
    so#73 72 9 so#73 77 9

ro#7 push                     (9)
rloc so#73 72 29
rclass cl#8
racs pub
rsig ty#2058
```

```
rlink C++
rstore NA
rvirt no
rtempl te#566
rcall ro#32 no so#73 74 17
rcall ro#33 no so#73 76 21
rpos so#73 72 9 so#73 72 52
    so#73 73 9 so#73 77 9

ro#32 isFull                  (10)
rloc so#73 27 29
rclass cl#8
racs pub
rsig ty#2054
rlink C++
rstore NA
rvirt no
rtempl te#562
rcall ro#31 no so#73 29 43
rpos so#73 27 9 so#73 27 43
    so#73 28 9 so#73 30 9

ty#5 int                      (11)
ykind int
yikind int

cl#8 Stack<int>               (12)
cloc so#66 23 15
ckind class
ctempl te#559
cfunc ro#6 so#73 7 24
cfunc ro#8 so#73 17 29
cfunc ro#32 so#73 27 29
cfunc ro#766 so#66 30 28
cfunc ro#767 so#66 32 18
cfunc ro#768 so#66 33 18
cfunc ro#7 so#73 72 29
cfunc ro#9 so#73 85 31
```

```
cmem theArray
cmloc so#66 38 28
cmacs priv
cmkind var
cmtype cl#63
cmem topOfStack
cmloc so#66 39 28
cmacs priv
cmkind var
cmtype ty#5
cpos so#66 23 9 so#66 23 19
    so#66 24 9 so#66 40 9

ty#9 bool                     (13)
ykind bool
yikind char

ty#14 void                    (14)
ykind void

ty#49 const int &             (15)
ykind ref
yref ty#439

ty#439 const int              (16)
ykind tref
ytref ty#5
yqual const

ty#2054 bool () const         (17)
ykind func
yrett ty#9
yqual const

ty#2058 void (const int &)    (18)
ykind func
yrett ty#14
yargt ty#49 F
```

Figure 3. Excerpts from the PDB file for the `Stack` code. The header file `StackAr.h` (`so#66` at (2)) “includes” the implementation file `StackAr.cpp` (`so#73` at (5)), so that templates are instantiated in the PDB file. These files define the class template `Stack` (`te#559` at (7)). (Strings containing template definitions have been partially deleted here.) The `Stack<int>` class (`cl#8`) instantiates `te#559`. Attributes and members of the class are given at (12). The `Stack<int>` function `push()` (`ro#7`) instantiates a function template (te#566 at (8)). Attributes of `push()`, and the routines it calls, are specified at (9). The function signature (`ty#2058`) reveals return and parameter types at (18).

specifies routine 7. The subsequent sequence of lines specifies the values of pertinent characteristics: for example, attributes of a routine include parent class, signature, the template from which it was instantiated, and the routines it calls. Attributes for the different item types are summarized in Table 1.The program database is stored in a relatively compact and portable ASCII format. Figure 3 shows excerpts from the PDB file for the templated `Stack` code in Figure 1.

3.3 DUCTAPE Library

DUCTAPE is a C++ library that provides an object-oriented API to PDB files produced by the IL Analyzer. Each item type of the PDB format is represented by a class having a corresponding name. All information about these items is accessible through member functions of the DUCTAPE classes. Common attributes were factored out into generic base classes, resulting in the class hierarchy shown in Figure 4. The root class of the hierarchy is `pdbSimpleItem`. `pdbSimpleItems` have two attributes, their name

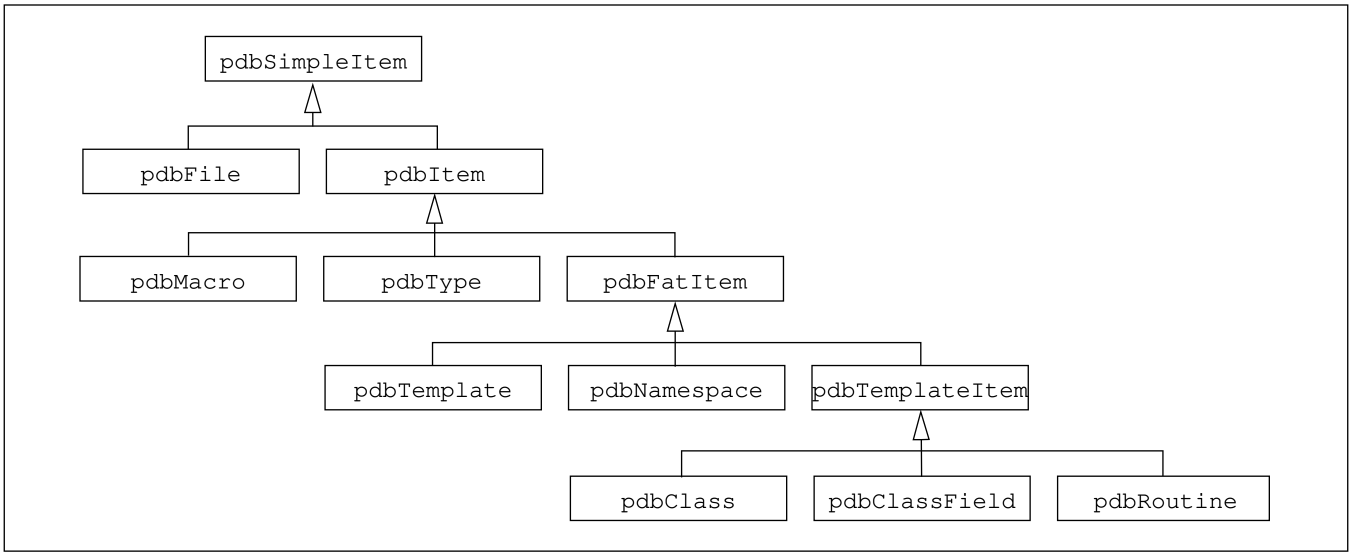

Figure 4. DUCTAPE Class Hierarchy

and PDB ID. Derived from `pdbSimpleItems` are `pdbFiles` and more complex `pdbItems`, which have a source code location, possibly a parent class or namespace, and an access mode. `pdbItems` are `pdbMacros`, `pdbTypes`, or so-called "fat" items. `pdbFatItems` have a header and a body, and attributes describing the source location of these parts. `pdbFatItems` include `pdbTemplates`, `pdbNamespaces`, and `pdbTemplateItems`. `pdbTemplateItems` are entities that can be instantiated from templates. The internal base classes are useful in DUCTAPE application programs when heterogeneous lists of items must be processed or stored (for example, `list<pdbTemplateItem>` can store a list of all template instantiations).

In addition, there is a class `PDB` that represents an entire PDB file. It provides methods to read, write, and merge PDB files, and to get the source file inclusion tree, the static call tree, and the class hierarchy. It provides a list of all items contained in the PDB file as well as lists of all defined types, files, classes, routines, templates, macros, and namespaces. Attributes of items representing references to other entities are implemented in DUCTAPE by pointers to the corresponding objects, allowing easy navigation through the available program information.

With the DUCTAPE library, PDT provides useful static analysis tools -- **pdbconv**, **pdbhtml**, **pdbmerge**, and **pdbtree**. Their functionality is summarized in Table 2. These applications also serve as examples of programming with the DUCTAPE library. The primary routine that **pdbtree** uses to display call graphs, for example, is given in Figure 5. In relatively few lines of code, a tool of some complexity was easily implemented using the DUCTAPE API.

4 Applications Using PDT

To demonstrate PDT's range of utility and ease of use, we report on two different applications -- the TAU (Tuning and Analysis Utilities) framework [17] and the SILOON (Scripting Interface Languages for Object-Oriented Numerics) toolkit [16] -- that uti-

Table 2: DUCTAPE Utilities

Utility	Functionality
pdbconv	converts files in the compact PDB format into a more readable format
pdbhtml	automatically creates web-based documentation that enables navigation of code via HTML links
pdbmerge	merges PDB files from separate compilations into one PDB file, eliminating duplicate template instantiations in the process
pdbtree	displays file inclusion, class hierarchy, and call graph trees

```
static void printFuncTree(const pdbRoutine *r, int level) {
    r->flag(ACTIVE);
    pdbRoutine::callvec c = r->callees( );
    for (pdbRoutine::callvec::iterator it=c.begin( ); it!=c.end( ); ++it) {     (1)
        const pdbRoutine *rr = (*it)->call( );
        if ( level != 0 || rr->callees( ).size( ) ) {
            cout << setw((level-1)*5) << "";
            if ( level ) cout << "`--> ";
            cout << rr->fullName( );                                             (2)
            if ( (*it)->isVirtual( ) ) cout << " (VIRTUAL)";
            if ( rr->flag() == ACTIVE ) {
                cout << " ..." << endl;
            } else {
                cout << endl;
                printFuncTree(rr, level+1);                                      (3)
            }
        }
    }
    r->flag(INACTIVE);
}
```

Figure 5. Source code from DUCTAPE's **pdbtree** utility displays the static call graph. The `for` loop (at (1)) iterates over functions called by the current function, reporting them (at (2)) as well as the functions that they call, recursively (at (3)). Functions instantiated from templates are automatically included in the vector of called functions.

lize PDT. TAU uses PDT to generate information needed to automatically instrument C++ source code. SILOON uses PDT in the generation of glue and skeleton code required in providing scripting language access to scientific libraries.

4.1 TAU Performance Profiling

TAU provides performance instrumentation, measurement, and analysis tools for the C/C++, Fortran, and Java languages. Its profiling and tracing toolkit currently uses PDT for automatic instrumentation of C++ source code. The TAU instrumentor iterates through the PDB descriptions of functions and templates to rewrite the original source file, annotating the functions with TAU measurement macros. The translated source code can subsequently be compiled and linked with the TAU library. When the executable is run, complete run-time statistics are collected, analyzed, and displayed via TAU.

Templates posed several problems in source-level instrumentation. A source-level instrumentation strategy for templates that is portable and independent of compiler instantiation schemes requires generation of a unique string for each template instantiation that identifies, if possible, the type information of the template parameters and return value along with the template name. TAU accesses C++ type information for instantiated templates at run time via the `CT(obj)` macro, which returns a string containing the type of the object `obj`. For each template, TAU determines if the given routine belongs to a class and that it is not a static member function, as shown in Figure 6. If these conditions are satisfied, then TAU inserts `CT(*this)`, which returns the type of the object with which the member function is associated. The unique instantiation of the class can therefore be incorporated in the name of an instantiated template, as in:

```
template<class T>
class vector {
public:
vector(int size) {
   TAU_PROFILE("vector::vector( )",
      CT(*this), TAU_USER);
   ...
   }
}
```

The Program Database Toolkit has been in use by TAU since early fall 1998, when it was applied to the POOMA (Parallel Object-Oriented Methods and Applications) [5] framework. POOMA uses templates extensively to provide array-related algorithms and manage allocation of system and network resources. Using PDT's predecessor (Sage++ [9]), automatic instrumentation of POOMA code had been attempted with TAU, but difficulties were encountered in parsing POOMA's complicated template entities. PDT's use of

```
// Get the list of templates.
PDB::templatevec u = pdb.getTemplateVec();
for(PDB::templatevec::iterator te = u.begin(); te != u.end(); ++te) {              (1)

    if ( (*te)->location().file() == file) {
        pdbItem::templ_t tekind = (*te)->kind();
        if ((tekind == pdbItem::TE_MEMFUNC)                                         (2)
            (tekind == pdbItem::TE_STATMEM) ||
            (tekind == pdbItem::TE_FUNC)) {

            // Templates need some processing.
            // The target helps identify if we need to put a CT(*this) in the type.
            if ((tekind == pdbItem::TE_FUNC) || (tekind == pdbItem::TE_STATMEM)) {  (3)
                // There's no parent class. No need to add CT(*this).
                itemvec.push_back(new itemRef(*te, true));

            } else {
                // It is a member function, so add CT(*this) via "false" argument
                itemvec.push_back(new itemRef(*te, false));
            }
        }
    }
}
sort(itemvec.begin(), itemvec.end(), locCmp);
```

Figure 6. Using PDT, the TAU instrumentor ascertains that a template is a class member function before using run-time type information (RTTI). The `for` loop iterates over all templates (at (1)). The `if` condition filters out non-function templates (at (2)). The `if-else` statement specializes the processing of member and non-member functions (at (3)).

the EDG Front End eliminated the C++ parsing problems. Figure 7 shows profile displays of time spent in POOMA's Krylov Solver routines that were generated with TAU automatic instrumentation.

4.2 SILOON

SILOON provides scientists with toolkits and run-time support for building easy-to-use external interfaces to existing high-performance libraries. The external interfaces are orchestrated via scripting languages to create domain-specific problem-solving environments. To achieve this, SILOON automatically generates bridging code that allows users to interface Perl and Python scripts with C++ libraries, as shown in Figure 8.

The SILOON toolkit uses PDT to parse source code from existing object-oriented class libraries and extract information regarding the interfaces to functions and class methods. This information is then used to generate bridging code, which, when compiled, provides the run-time support for linking user scripts with back-end computational engines. The code generation builds language-specific wrapper functions and language-independent bridging code. The wrapper functions are written in the scripting language, and provide a natural and convenient interface to the C++ library. The wrapper functions call the lower-level bridging functions written in C++ and accessible across all scripting languages. These functions register user-designated library routines with SILOON's routine management structures, and process function calls from the scripting languages.

With PDT, users simply give their C++ source code as input to SILOON, rather than specify their interfaces in an interface definition language (IDL). While there are applications similar to SILOON (SWIG [8] being the most well-known), none offers the same level of support for C++. Because PDT uses an ANSI-compliant C++ parser, SILOON is able to handle many of the complexities of C++ correctly, including:

- templated classes and functions,
- virtual and static member functions,
- constructors and destructors,
- overloaded operators and functions,
- default function arguments,
- references,
- enumerations,
- `typedefs`, and
- the Standard Template Library (STL).

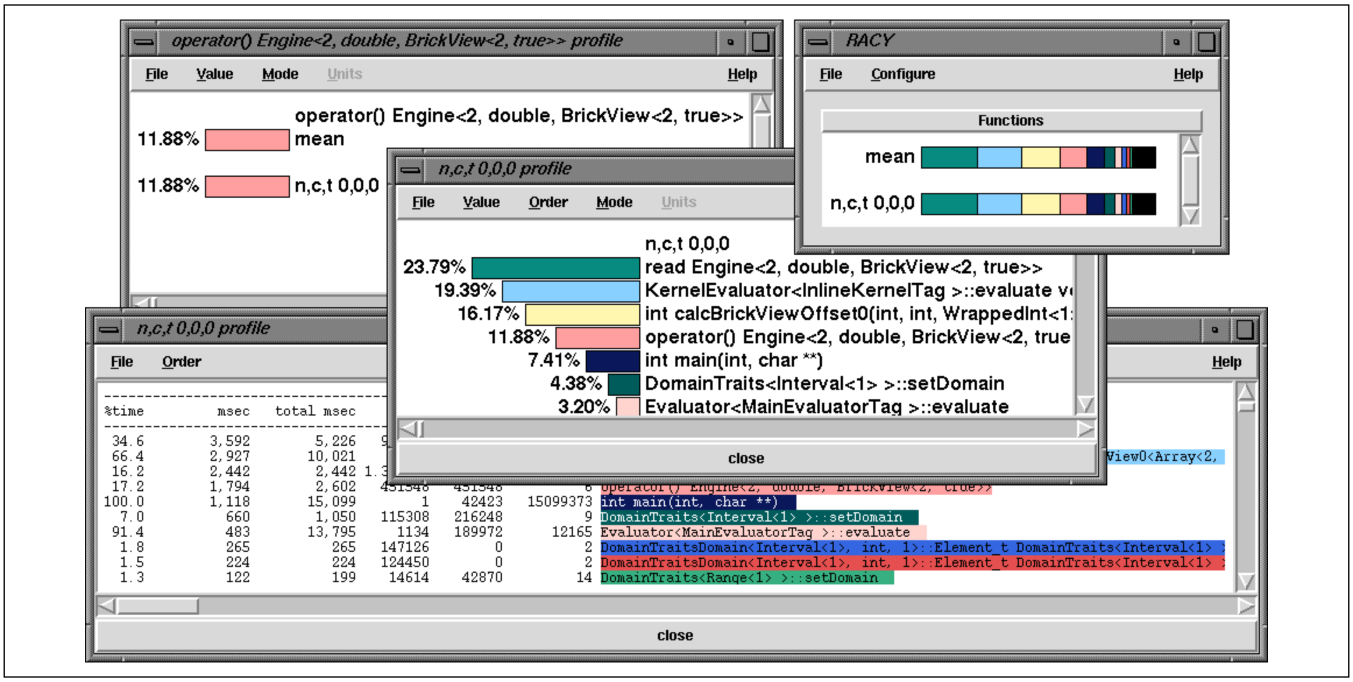

Figure 7. TAU automatically instrumented POOMA's Krylov solver using PDT.

Without PDT's capabilities, many of the above features would not be available in SILOON.

Templates are treated the same as other entities by SILOON, with the exception that non-alphanumeric characters in the name are mangled (*i.e.*, transformed to include information on types and qualifiers), so that they can be accessed in scripting languages. To handle templates in SILOON, it is necessary to instantiate and compile into the SILOON library any templates that the user wants to be available. Currently, the user must explicitly instantiate such templates in the parsed code; only these instantiations are included in PDT's output. A useful extension to PDT would be to provide access to all templates, whether instantiated or not. SILOON could then present a template list to the user, and automatically generate instantiations of selected templates.

5 Related Work

Other systems that concentrate on high-level language interfaces in software development exist. The four with a focus most similar to the Program Database Toolkit are discussed here. Sage++ [9] is an object-oriented compiler toolkit that assists in constructing source-to-source translation tools. It uses a three-step process: source code is parsed, the parse tree restructured, and the restructured tree unparsed. The Sage++ class library enables construction and insertion of profiling objects at the beginning of routine structures during the second step. It was used in previous versions of TAU, but does not adequately support templates. The ASTLOG [10] language was developed for an analysis and debugging tool. Like the IL Analyzer, this tool extracts high-level interfaces from C++ code, accessing the syntax tree via user-defined node traversal and pattern matching predicates, and accumulating query results using the underlying set predicates of Prolog. Unfortunately, the ASTLOG tool does not provide information about source code locations. In the Concert [7] distributed system, compilation produces an intermediate representation that is an interface definition language. Programmers annotate C or Fortran code (without templates), enabling the Concert front end to derive interfaces that become input for two back ends, a stub compiler and an interpreter that "generate" marshalling and unmarshalling of message signatures. The resulting interfaces ensure the interoperability of different languages in a distributed setting, and can describe implementation-related information, such as order and memory layout of parameters. SUIF (Stanford University Intermediate Format) [20] and its extension OSUIF (Object SUIF) [11] provide compiler infrastructure toolkits that enable optimization.

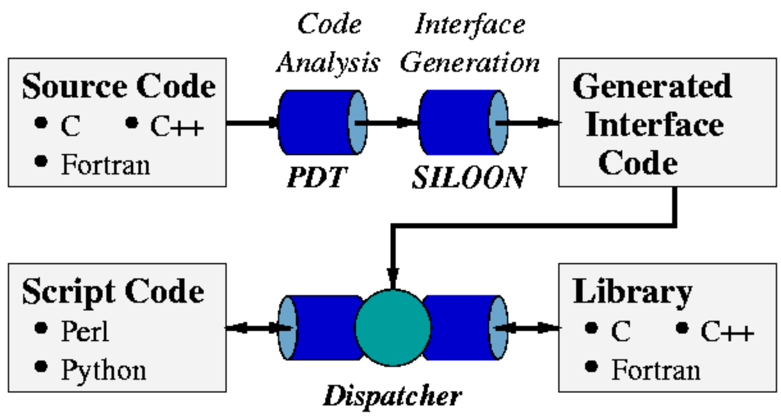

Figure 8. Using PDT, SILOON automatically generates code that links scripting languages with user libraries.

Whereas SUIF's goal was development of techniques for parallelization, OSUIF's focus is on compilation of object-oriented languages. In OSUIF, high-level information on object-oriented source language constructs is accessible. However, OSUIF efforts are not aimed at providing an infrastructure for tool development in the manner of PDT. All of these systems had limitations precluding their use in complex applications: limited support for recent features of C++, lack of source location information, manual specification of interfaces, or lack of an appropriate infrastructure for tools.

6 Conclusions and Future Work

Version 1.3 of the Program Database Toolkit for C++ has been released [3]. The distribution includes the C++ Front End, the IL Analyzer, and DUCTAPE, all of which process templates and instantiations. In addition, the TAU performance instrumentation tool, SILOON analysis support, and various PDT processing tools (**pdbmerge**, **pdbconv**, **pdbtree**, and **pdbhtml**) are available for use with PDT 1.3. All handle template entities. The inclusion of KAI's [13] 3.4c standard library header files has significantly improved PDT's robustness of parsing and analysis, while increasing the scope of supported platforms and simplifying configuration.

Support for multiple programming languages is crucial in the high-performance environments in which the Program Database Toolkit is used. We plan to extend PDT's scope to support the Fortran 90 and Java languages. The challenge is to determine where the C++ PDB constructs can be reused, and where they must be extended to provide language-specific support. For instance, TAU must know the locations of Fortran routine entry and exit points to insert profiling instrumentation. A Fortran 90 IL Analyzer is currently being implemented,[1] and the structure of the program database modified, to handle Fortran 90's constructs. Fortran derived types and modules will correspond to C++ classes/structs/unions, while Fortran interfaces will correspond to routines with aliases. Fortran array features will be specified with new attributes. DUCTAPE will be enhanced to accommodate these changes to the program database. We are also planning to develop a Java IL Analyzer based on EDG's Java Front End, with the PDB and DUCTAPE enhanced to accommodate Java's constructs. In general, if the Program Database Toolkit can make a language-specific parse tree accessible in a uniform manner, static analysis tools and other applications can be built that process different languages in a uniform and consistent way.

7 Acknowledgments

This work has been supported by DOE2000 grant #DEFC0398ER259986 and ASCI Level 3 grant #03588-001-994R from the Department of Energy.

---

[1] This IL Analyzer is derived from the Fortran 90 Front End distributed by Mutek [15], which was based on the Fortran 77 Front End distributed by EDG [12].

8 Bibliography

[1] J. S. Adamczyk and J. H. Spicer. Template Instantiation in the EDG C++ Front End. Edison Design Group Technical Report, 1995.

[2] Advanced Computing Laboratory/LANL. ACL Research. http://www.acl.lanl.gov/research/, 1999.

[3] Advanced Computing Laboratory/LANL. PDT: Program Database Toolkit. http://www.acl.lanl.gov/pdtoolkit/, 1999.

[4] Advanced Computing Laboratory/LANL. PDT: Program Database Toolkit. Supercomputing '99 flyer, Los Alamos National Laboratory Publication, LALP-99-204, November 1999.

[5] Advanced Computing Laboratory/LANL. POOMA: Parallel Object-Oriented Methods and Applications. http://www.acl.lanl.gov/pooma/, 1999.

[6] ASC X3. International Standard: Programming Languages - C++. ISO/IEC 14882. Information Technology Council (ITI), 1998.

[7] J. S. Auerbach and J. R. Russell. The Concert Signature Representation: IDL as Intermediate Language. SIGPLAN Notices, vol. 29, no. 8, pp. 1-12, August 1994.

[8] D. M. Beazley. SWIG: Simplified Wrapper and Interface Generator. http://www.swig.org/, 1998.

[9] F. Bodin, P. Beckman, D. Gannon, S. Narayana, S. Srinivas, and B. Winnicka. Sage++: A Class Library for Building Fortran 90 and C++ Restructuring Tools. Proceedings OONSKI94, the Second Annual Object-Oriented Numerics Conference, pp. 122-138, April 1994.

[10] R. E. Crew. ASTLOG: A Language for Examining Abstract Syntax Trees. Proceedings of the Conference on Domain-Specific Languages, pp. 229-242, October 1997.

[11] A. Duncan, B. Cocosel, C. Iancu, H. Kienle, R. Rugina, U. Hoelzle, and M. Rinard. OSUIF: SUIF 2.0 with Objects. Second SUIF Compiler Workshop, August 1997.

[12] Edison Design Group. Compiler Front Ends for the OEM Market. http://www.edg.com/, 1998-1999.

[13] Kuck & Associates, Inc. KAI. http://www.kai.com/, 1999.

[14] K. Lindlan, J. Cuny, A. D. Malony, S. Shende, and P. Beckman. An IL Converter and Program Database for Analysis Tools. Proceedings of 2nd SIGMETRICS Symposium on Parallel and Distributed Tools, p. 153, August 1998.

[15] Mutek. Fortran 90 Front End Documentation. http://www.mutek.com/, 1999.

[16] R. D. Rivenburgh, C. E. Rasmussen, K. A. Lindlan, B. Mohr, and P. H. Beckman. Automatic Generation of Perl Extensions to C++ and Fortran 90 Class Libraries. O'Reilly Open Source Software Convention, July 2000.

[17] S. Shende, A.D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel, Scientific Applications Using C++. Proceedings of 2nd SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 134-145, August 1998.

[18] B. Stroustrup. The C++ Programming Language, Third Edition. Addison-Wesley, 1997.

[19] M. A. Weiss. Data Structures and Algorithm Analysis in C++. Benjamin Cummings, 1994.

[20] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessey. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. ACM SIGPLAN Notices, vol. 29. no. 12, pp. 31-37, December 1994.