

Beyond ASIS: Program Data Bases and Tool-Oriented Queries

Janusz Laski
SofTools, Inc.
3024 Longview
Rochester Hills, Michigan 48307
(248) 853 7602
laski@oakland.edu

William Stanley
Oakland University
Rochester, Michigan 48309
(248) 370 2133
wfstanle@oakland.edu

Pawel Podgorski
WorldClass Technology
30200 Telegraph Road, Suite 401
Bingham Farms, Mi 48025
(248) 646 0034/230
Pawel_Podgorski@wct.com

1. ABSTRACT

The availability of higher level ASIS libraries is of prime importance for the real success of ASIS technology to facilitate the development of Software Analysis and Testing (SAT) tools. This is due to the fact that ASIS queries are expressed in terms of the syntax of an immensely complex language and do not directly support the objectives of a tool builder. In this paper we discuss two plausible sets of higher levels, tool-oriented queries about the Program Under Analysis (PUA), which ideally do not require the knowledge of ASIS. We first present our experience with using ASIS to develop the front end of SWAT (SoftWare Analysis and Testing), a comprehensive system for static and dynamic (execution-based) software analysis. In it, ASIS is used to retrieve the information about the PUA and store it in Program Data Bases (PDBs). Then, an SAT tool uses the PDBs rather than ASIS to get the necessary information about the PUA. We hypothesize that the access functions to the PDBs can be viewed as second-level queries about the PUA;

if well thought-out, a claim not made here, those queries can be used to build a wide class of program browsers. If, however, a more complicated tool is needed, a third level of queries can be specified. We illustrate that possibility by briefly showing how SEER, SWAT's dependency analyzer, uses several PDBs to carry out the dependency analysis whose results are stored in the *derived* PDBs; they are produced by processing the data in the original, *primary* PDBs. Again, the relations stored in the secondary PDBs gives rise to the third level queries about the PUA. It is obvious that higher level queries neither have to be implemented by an underlying ASIS layer nor the PDBs have to be used for that purpose. The main motivation here was to do that analysis *without a direct* use of ASIS. Indeed, we stress the importance of research into various SAT tools that would provide guidance in the definition of an "optimal" query language. In particular, we emphasize the need for ASIS queries that support dynamic, execution-based program analysis.

1.1 Keywords

Software, verification, testing, static analysis, dynamic analysis, program dependencies, Ada, ASIS, program data bases, queries.

2. INTRODUCTION

There have been suggestions around to define a second level of ASIS queries as a support for the development of Software Analysis and Testing (SAT) tools. Such an approach would if not only eliminate then at least alleviate

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.
To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGAda 2001 09/01 Bloomington, MN, USA
© 2001 ACM 1-58113-392-8/01/0009...\$5.00

the difficulty of using ASIS directly. In this report we address the issue by generalizing our experience with the development of SWAT, SoftWare Analysis and Testing, a comprehensive system for static and dynamic (execution-based) software analysis.

Here is the organization of the paper. In Section 3 the scope of SWAT's functionality and the difficulties of using ASIS are illustrated. In Section 4 the (Primary) Program Data Bases (PDBs), both at system and procedural levels, produced by SWAT to store the ASIS-retrieved information about the Program Under Analysis (PUA) are described. The PDBs gives rise to potential Second-Level ASIS queries. Those queries support a wide class of browsers of the PUA. In Section 5 we discuss the general requirements of tools that support semantic, rather than clerical, programming activities. We illustrate those by discussing two functions of SEER, the SWAT's static analyzer. The results of static analysis are stored in the *derived* PDBs; those are the product of processing the primary PDBs. Again, we suggest that those results may be generalized as the third level queries about the PUA. Finally, in Section 6 future directions are discussed. Specifically, we stress the importance of dynamic analysis and the need for suitable ASIS queries.

Throughout, an example program in Appendix is used to illustrate the discussion.

3. ASIS AND SWAT

In this Section we discuss the main philosophy behind SWAT's design and the problems encountered while using ASIS to develop the front end of the system.

3.1 Scope of SWAT

SWAT an experimental system for testing the power of various SAT methods and tools and facilitate research into new methods. Its two main design premises are *integration* and *extensibility*. The former means that the SAT tools should be centered around a common conceptual framework; it also means that dynamic (execution-based) analysis should be an extension of static analysis. Extensibility means that it should be possible to incorporate new SAT methods using the existing front end.

The initial static functionality of SWAT includes the derivation of Control and Data Dependencies [3,10], support for Program Slicing (*partial programs* [2]), and the identification of data and control anomalies. Among dynamic methods SWAT is to support structural (White-Box) testing (statement, branch and data flow coverage, cf [9]) and Dynamic Slicing [8]. However, those were only the initial requirements. It was expected that ongoing research

would result in new methods which will have then to be incorporated into SWAT. Therefore, the open-ended structure of the system was of paramount importance. Consequently, the front end was to provide information about the program that would be sufficient to support a wide class of SAT methods.

To support that paradigm we used our earlier experience with the development of STAD (System for Testing And Debugging [9]) to retrieve and store information about the PUA in a collection of files referred to as Program Data Bases (PDBs), described in some detail in Section 4. At present, the PDBs do not conform to the structure of relational data bases as, for example, do those in [12]. Clearly, the PDBs, once created do not change and therefore most of the operations of relational data bases are really not needed. What is important here is not the particular form of the PDBs but their *contents*, that is, what information should they contain to facilitate the development of a variety of SAT tools *without* the necessity of using ASIS.

Only after the PDBs have been designed did we realize that the information stored in them might be used to define second-level queries about the PUA, as discussed in Section 4.3.

3.2 Using ASIS

The Ada Semantic Interface Specification (ASIS, [7]) is a set of Ada packages that provides collections of queries about an Ada Program Under Analysis. Although our experience confirms that ASIS *is* a great help in the development of a front-end, that does not mean that it is the programmer's paradise, either. The main reason for that is the fact that ASIS queries are expressed in terms of the syntax of an immensely complex language, rather than in those of a tool-builder. It is due to this close relationship between ASIS and Ada syntax that the Ada Language Reference Manual (LRM) should become the ASIS programmer's best friend; the ASIS Standard itself should be a close second. This fact creates serious difficulties for tool builders who, typically, think in tool-oriented terms, rather than in syntactical ones.

To illustrate the problem, consider the derivation of the flow graph for the simple IF statement in Figure 1.

```
IF A OR ELSE B THEN
  Integer_Variable := 1 + 2;
END IF;
```

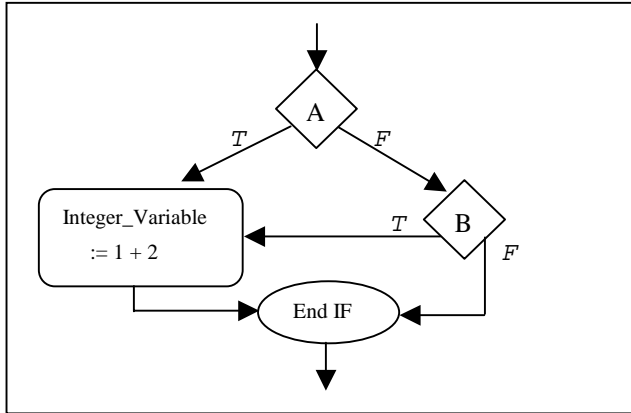


Figure 1. The Flow Graph of an IF statement.

We shall now show how ASIS can be used to break the above statement into nodes for the flow graph. Note there are many other issues involved with building the flow graph that are too involved to cover here. That applies, in particular, to the identification of the arcs and their labels - True, False for IF and loop statements, case expressions for CASE statements and Null for sequential transfer of control.

First, we use ASIS queries to gather the declarations of individual routines (beyond the scope of this example). These declarations are returned in the form of an Asis.Element, which can be thought of as a pointer to the abstract syntax tree. Then the control flow graph builder uses ASIS queries to further break down the declarations and build the flow graph.

The following pseudocode illustrates this approach – the bullets (•) mark the continuation of the preceding line.

```
ProcedureBreak_Into_Node
  (Routine_Declaration: Asis.Element) is
--Break declaration into a list of its
-- individual statements and
-- a list of declarations
Routine_Statement_List:
• Asis.Statement_List :=
• Asis.Declarations.Body_Statements
• (Routine_Declaration);
```

```
begin
  For index in
    Routine_Statement_List'Range loop
    IF Asis.Elements.Statement_Kind
      (Routine_Statement_List(index))
      = An_If_Statement THEN
--we will just look for the IF statement
    • kind
-- Break the if statement into its
-- different paths with the ASIS query:
    Asis.Statements.Statement_Paths
    • (Routine_Statement_List(index),False);

-- Determine the path kind, and if
-- appropriate get the conditional
-- expression of the path using the
-- query:

    Asis.Statements.Condition_Expression
    • (If_Paths(Index));

-- The conditional expression is then
-- passed to another handler which
-- determines whether it is a short-
-- circuit expression, and builds nodes
-- for the expressions accordingly. Then
-- the paths are broken into their
-- respective statements using the
-- query:

    Asis.Statements.Sequence_Of_Statements
    • (If_paths(Index));

-- Each statement is then evaluated and
-- processed.
    End if;
    End loop;
end Break_Into_Nodes;
```

This is obviously oversimplified, but the format is correct. It is obvious that the tool builder should be able to derive the flow graph in a less painful way. This is where our Data Bases come to the rescue; one can use them as a "second layer" of ASIS queries, without having to learn ASIS.

4. PROGRAM DATA BASES.

The Program Data Bases produced by SWAT are actually ASCII text files whose format was designed to be both readable by people and easily accessed by tool-builder programs. Since their structure was specified early on, a parallel development of the front end and the tool proper (see Section 5) was possible.

The Data Base files are divided into System Data Bases and Procedure-Level Data Bases. System-Level Data Bases, in general, represent the highest-level views of the PUA. A

system level Data Base would contain information that spans an entire program rather than a single subprogram. In contrast, a Procedure-Level Data Base contains information about individual subprograms.

In the rest of the section we briefly discuss the most important (from the point of view of a tool builder) Program Data Bases (PDBs) produced by our Ada95 Front End and propose a corresponding query language. The example PDBs refer to procedure Home_Budget in Appendix.

4.1 System Data Bases

SWAT produces the following System Data Bases:

Elaboration Order (PUA.eor), which lists all available compilation units in the program in the implementation-defined elaboration order.

With Relation (PUA.wth), which identifies the With dependency in the program.

Routine Call Graph (PUA.rcg): For each subprogram R, subprograms potentially called by R are identified, cf [5]. The calling and called subprogram are identified by both their name and an index to the Routine Symbol Table (RST), see Figure 2. It also indicates the nodes for the flow graph (in the calling subprogram) where the subprogram calls can be found. The flow graphs can be found in the Control Flow Graph (CFG) databases.

In the example in Figure 2, the number 4 that refers to a call to Get_Expenses is the index of a block in the Routine Symbol Table. The lines in question have been underlined to illustrate the relationship. Throughout the SWAT databases there are similar cross references. The indices are roughly analogous to pointers. Parts of the databases are omitted because of space constraints, the missing text is denoted by five hyphens (-----).

Routine Symbol Table (PUA.rst), which provides an external view of each procedure and function in the program. That view includes: types and modes (in, in out, out) of the formal parameters, global variables manipulated and their modes, and the nesting tree of the routines. *Declaration blocks* and *FOR loops* are treated as separate routines since they introduce local variables. *Package specification* (with possible variable initialization) and *body* are also treated as separate routines. When FOR loops and declaration blocks have no name associated with them, SWAT-defined names are given to them to allow the user to refer to them by name, e.g. \$For2. Cross-references to the source and other PDBs are included.

For example, when the entry for Get_Expenses in Figure 3 is examined, we find the following information:

This is the fourth entry in the RST, thus the index = 4.
 The full name is "Home_Budget.Get_Expenses"
 The parameters are "(Budget: OUT Budget_Array)"
 The parameter Budget is the third entry in the VST.
 Since this is a procedure there is no returned value.
 The subprogram does not use any global variables.

This is only some of the information that can be gathered from the RST only. Since indices to other PDBs are in the RST, more information can be gathered by examining these PDBs.

Type Symbol Table (PUA.tst): Each type that appears in the program, whether standard or user-defined, is listed with its unique ID number. Anonymous types ("on the fly" type declarations without type names associated with them) are given system names and are included in the table.

Variable Symbol Table Data (PUA.vst): Each variable in the program is given its fully qualified name, a unique integer ID, and cross references to the Routine Symbol Table, Type Symbol Table, and Index Data Bases.

Visibility (PUA.vis): For every entity in the program (type, variable, subprogram) its visibility and scope are identified.

More information on those is available at the site www.softools.org. Below we show the structure of some of them.

```
Routine Call Graph      Home_Budget.RCG
• 3  NUMBER OF ROUTINES
*****
1 IDENTIFIER NUMBER OF CALLING ROUTINE
•
  Home_Budget CALLS
2 NUMBER OF ROUTINES CALLED

2 IDENTIFIER NUMBER OF CALLED ROUTINE
•  Display NAME OF ROUTINE CALLED
4 6 0 NODES WHERE CALLED

4 IDENTIFIER NUMBER OF CALLED ROUTINE
•  Get_Expense NAME OF ROUTINE CALLED

5 0 NODES WHERE CALLED
-----
END FILE  Home_Budget.RCG
```

Figure 2. Part of Routine Call Graph for HomeBudget.

```

Routine Symbol Table      Home_Budget.RST
Routine_ID Kind_Code Parent_Code
• Routine_Name
  Qualifier_String
  Parameter_String
  Return_Type_Code Return_Type
  Number_of_Parameters Parameter_Codes
• Parameter_Types
Number_of_Global_Variables Variable_ID's
• Use/definition_Codes
6 = Number of routines or blocks.

-----

*****
4 2 1 Procedure Get_Expense NOT_EXPORTED

Home_Budget.
(Budget : out BudgetArray)
0 NULL
1 = Number of parameters
• 3 = Parameter Codes
• 4 = Parameter Types.
0 = Number of global vars
• 0 = variable ID's
• 0 = use/define codes
*****
5 4 4 FOR_LOOP $For2 NOT_EXPORTED
Home_Budget.Get_Expense.
NULL
0 NULL
0 = Number of parameters
• 0 = Parameter Codes
• 0 = Parameter Types.
0 = Number of global vars
• 0 = variable ID's
• 0 = use/define codes

END Routine Symbol Table

```

Figure 3. Routine Symbol Table: Entries for Get_Expenses and FOR loop.

```

Variables Symbol Table  Home_Budget.VST
FIELD DESCRIPTIONS:
Line 1: Object ID Number, Object name
Line 2: Cross reference to block in
• RST, Fully qualified name
Line 3: Cross reference to type in
• TST, type name, object kind
Line 4: Cross reference to file in
• IDX, first line, first column,
• last line, last column

9 = Number of Variables
*****
1 Max_Expense
  1 Home_Budget.Max_Expense
  1 Float CONSTANT NOT_EXPORTED
  1 6 4 6 14
*****
2 Budget
  1 Home_Budget.Budget
  4 BudgetArray VARIABLE NOT_EXPORTED
  1 51 3 51 8
*****
3 Budget
  2 Home_Budget.Display.Budget
  4 BudgetArray PARAMETER
• NOT_EXPORTED
  1 15 23 15 28
-----

END Home_Budget.VST

```

Figure 4. Part of Variable Symbol Table

4.2 Procedural Data Bases

Procedural Data Bases produced by SWAT refer to individual subprograms. Data bases of certain type (e.g., Control Flow Graphs) that correspond to subprograms declared in the same package are grouped in a single Data Base file for that package. Data Bases for individual subprograms, i.e., not declared in a package, are grouped in an "orphan" Data Base file. Here are the most important procedural data bases.

Control Flow Graph (*.cfg, the asterisk stands either for a package name or for "orphans") contains control flow graphs for each subprogram. Note, that although FOR loops and DECLARE blocks are treated as separate routines in the Routine Symbol Table, they do not have separate flow graphs for they are embedded in the appropriate subprograms. Package specification (variable initialization) and body (variable initialization and initialization routine) have two distinct flow graphs.

Definition-Use (*.dfu) contains, for each subprogram, variables used (referenced) and defined (assigned a value) in each node (instruction) in the subprogram. A distinction is made between a *total* vs. *partial* definition of arrays, cf [10]. For example, if A is an array, then A(i) := <expression> is a partial definition of A, while A := (aggregate) is considered total.

Node Data Base (*.ndb) contains the type and contents (action) of every node in the control flow graph.

Position Data Base (*.pos) provides cross-references between nodes in the flow graphs and the source code.

In the remainder of this subsection we show parts of the Control Flow Graph, Definition-Use and Nodes files for procedure Display from Appendix.

```
Routine Flow Graphs      Orphans.CFG
3  NUMBER OF ROUTINES
*****
2 = Routine ID

Routine Name =
•   Display(Budget: BudgetArray)

13  NUMBER OF NODES

1  BRANCH NODES .. (BRANCH NODE,
•   NUMBER OF SUCCESSORS,
•   NUMBER OF PREDICATES)
5 2 -99

1  JOIN NODES ... (JOIN NODE,
•   NUMBER OF PREDECESSORS)
5 2

13  ARCS
1 2  NULL
2 3  NULL
3 4  NULL
4 5  NULL
5 6  TRUE
5 11 FALSE
6 7  NULL
7 8  NULL
8 9  NULL
9 10 NULL
10 5  NULL
11 12 NULL
12 13 NULL
13 0  NO LABEL -- EXIT
END FILE  Orphans.cfg
```

Figure 5. The Control Flow Graph for procedure Display.

```
Definition Use Data Base      Orphans.DFU
3  NUMBER OF ROUTINES
-----
*****
2 = Routine Number
Display(Budget: BudgetArray) ROUTINE NAM
13 = NUMBER OF NODES

3  NUMBER OF VARIABLES MANIPULATED
3 4 5
### END OF VARIABLE LIST

1  NODE #
1  NUMBER OF VARIABLES DEFINED
3  Total  VARIABLE NAME
Budget  VARIABLE NAME
0  NUMBER OF VARIABLES USED

2  NODE #
0  NUMBER OF VARIABLES DEFINED
0  NUMBER OF VARIABLES USED

-----

5  NODE #
0  NUMBER OF VARIABLES DEFINED
2  NUMBER OF VARIABLES USED
4  Total  VARIABLE NAME
$For1.C  VARIABLE NAME
5  Total  VARIABLE NAME
$For1.$BOUND  VARIABLE NAME

-----

END FILE  Orphans.dfu
```

Figure 6. Part of the Definition-Use file for procedure Display.

```

Nodes DataBase      Orphans.NDB
3  NUMBER OF ROUTINES
*****
-----

PROCEDURE Display(Budget: BudgetArray)

13 = NUMBER OF NODES

NODE NUMBER      NODE TYPE
NODE TEXT -- SOURCE

1  EN -- ENtry, Paremeter Evaluation
Display(Budget: BudgetArray) IS

2  ST -- Start execution
BEGIN

3  FI -- FOR loop initialization
[ $For1.C := Categories'FIRST ]

4  FI

[ $For1.$Bound := Categories'LAST ]

5  FT -- FOR loop termination test
[ $For1.C <= $For1.$Bound ]

6  PC -- Procedure Call
Category_IO.put (c);

7  PC
Put ( ": " );

8  PC
Put ( item => Budget(C), aft => 2, Exp=>
0 );

9  PC
New_Line;

10  FINC -- FOR loop INCRement
[ $For1.C := $For1.C'SUCC ]

11  EL -- End of FOR Loop
END LOOP [ $For1 ]

12  PC
New_Line;

13  EX -- EXit
END Display

-----

END FILE      Orphans.ndb

```

Figure 7. Node Data Base for procedure Display; the node numbers are identified in source code in Appendix.

4.3 Second Level Queries

The idea behind the data bases discussed in the preceding sections was to provide the basic information needed to develop a wide class of SAT tools. In general, several data bases have to be inspected to retrieve the data needed. Naturally, the simplest tools that can be developed are *program browsers* that provide selected views of the program. For example, if one wants to know the type and declaration place of variable, say Budget, one first searches the VST data base for an entry for Budget (there are actually three such entries) and selects the one of interest using the fully qualified name. Then, using the cross-reference links there to the RST and TST files, one gets, respectively, the routine where the variable is declared and its type. Finally, using the links to the POS file (not yet completed), one gets the source code positions of the type and variables declarations.

Now, if one wants to know where Budget is defined and used, one first consults the VIS(ibility) file to identify the scope and visibility of the variable. Using that information one consults the corresponding DFU data bases for subprograms in the scope of the variable.

It is apparent now that one can define a set of queries that capture that process. Consider, for example, the following type definitions:

```

Type Procedure_Type is record
  Name:           String;
  Procedure_Decl: Asis.Element;
  Procedure_Body: Asis.Element;
  Global_Vars:    Asis.Element_List;
  Global_Vars_VST: Integer_List;
  -- etc.
End record;

```

```

Type Var_Identifier_Type is record
  Name:           String;
  Variable_Decl:  Asis.Element;
  Variable_Type_Decl: Asis.Element;
  VST_Index:      Natural;
  Variable_Type_Index: Natural;
  -- etc.
End record;

```

```

Type Graph_Nodes is record
  Asis_Node:      Asis.Element;
  Asis_Predecessors: Asis.Element_List;
  Asis_Successors: Asis.Element_List;
  Successors:     Integer_List;
  -- etc.
End Record;

```

Now the following system level query

```
function Variable_Declaration
(Var_identifier: Var_identifier_type)
  return Procedure_identifier;
```

may be defined as returning the name of the subprogram in which `Var_Identifier` is declared.

As an example of a procedural level query one can define

```
Function Node_Successor
(Node: Graph_Nodes; Routine:
Procedure_identifier)
  return Graph_Nodes;
```

that returns an array of successors of `Node` in the flow graph for `Routine`.

Certainly, further analysis is needed to catalogue and possibly standardize a set of "optimal" queries; we briefly discuss that issue in Section 6. Having defined such a set one has to confront the issue of an effective implementation.

As stated earlier, our choice of data bases has been influenced by our previous experience and no claim is made here about its uniqueness, efficiency or completeness. Indeed, it is most likely the PDBs will have to be revised to efficiently support typical queries about the program. Moreover, the very need for the PDBs may be debatable; for instance, for queries like `Variable_Declaration` there won't be a need to derive the entire data bases. In contrast, to provide an answer to the `Node_Successor` query, one most likely will have to derive the entire flow graph for `Routine`. Moreover, for queries which do need major portions of databases (whether transparent to the user or not) one will have to decide their actual format, e.g. the choice between several smaller relations vs. one bigger relation.

5. DERIVED DATA BASES AND HIGHER LEVEL QUERIES

The PDBs and the corresponding second-level browser queries essentially support clerical, rather than semantic activities, cf [6]. However, published estimates suggest that the former constitute only about 20% of programming effort, with 80% falling into the latter category [4]. Unfortunately, it is much more difficult to support reasoning about the program than to retrieve information about it, however tedious that task may be. It is then not surprising that with a few notable exceptions (cf [1]) most SAT tools support clerical activities. However, one has to be prepared to develop new, more semantic-oriented tools

as the result of the ongoing basic research. Here is the way we address the issue in SWAT.

The PDBs in Section 4 are derived directly from the source code and hence are viewed as "primary," since they contain the raw, further unprocessed information about the program. A program browser would access several primary databases to get the information needed to answer a second-level query. In contrast, a semantic-oriented tool *processes* several primary PDBs to get the desired information. The results of the processing are stored in Secondary Data Bases and give rise to the *Third Level Queries*. We illustrate that approach on the Routine Sequence Graph, a system level query, and Definition-Use Chains, essentially identical to Data Dependency Graph [3], a procedure level query. Both queries are supported by SEER, SWAT's Static Analyzer.

Routine Sequence Graph (PUA.rsg) provides a finer view of the calling structure of the subprograms than the Call Graph (PUA.rcg) does. Clearly, in the RSG *every* call of a procedure is a node in the graph, with the surrounding environment of the caller abstracted to the relevant control flow (control dependence, cf [3]).

For example, the Call Graph for `Home_Budget` shows that the latter *may* call `Display` and `Get_Expense`. In contrast, the RSG identifies the following sequence of calls: `Display`; `Get_Expense`; `Display`;

The rsg allows one to identify all potential sequences of calls and, consequently, identify potentially invalid ones, such as `Create(Stack)`; `Pop(Stack)` (No `Push(Stack,item)` in between).

Let `P` and `Q` be routines in the program, `n` and `m` be, respectively, nodes in `P` and `Q`. There is a *Definition-Use Chain* (DUC) relation between the pairs (P,n) and (Q,m) if and only if there exists variable `v` in the program that is assigned a value in `n` and that value *may be* used in `m`. That means that there exists a path from `n` in `P` to `m` in `Q` along which the variable `v` is *not* redefined. The DUC relation is stored in the derived data base `proc.duc`. If, moreover, some variable `w` is defined in `m`, there is a *Data Dependency* from `n` to `m`, cf [3,9,10].

The Data Dependency is essential for reasoning about the program such as (static) debugging and regression analysis [11], while Definition-Use relation is the basis for Data Flow testing [9].

6. CONCLUSIONS AND FUTURE RESEARCH.

We have presented our experience with the use of ASIS to develop the front end of SWAT and its static analyzer,

SEER. We found ASIS to be enormously helpful but we believe that tool builders would be even better helped if they had at their disposal a system of queries that does not need the knowledge of ASIS. This is of particular importance since most of the research into the Software Analysis and Testing is *language independent*. Towards that goal we proposed two higher-level sets of queries, as a generalization of Program Data Bases used in SWAT: The program browser supporting queries and ones that support semantic program analysis.

No claim has been made that our approach offers an ultimate solution to the higher-levels ASIS problem. Naturally, more research is needed to define queries that support the development of any conceivable SAT tool. At the current state of the art it does not seem plausible to expect an agreement about the semantic-oriented queries. Clearly, at this point no particular subset of semantic STA methods can be strongly recommended since (with the exception of formal proving and anomaly detection) we don't yet know what we know. Indeed, all claims about the magical power of this or that method should be taken with a solid grain of salt; when the laboratory results are scaled up to the real world size, the negative results most likely will get worse, while the positive ones will still have to be field tested.

However, it does seem possible to agree on the second level of ASIS queries or, more appropriately, the *first level of tool oriented queries* since, in principle, the developer might choose to use directly the Abstract Syntax Tree rather than ASIS, cf SPARK [1]. As a rule of thumb, such a set of queries should (1) answer any question about the PUA that does *not* involve either a further processing of the raw information about the program or the program's actual execution, (2) allow the insertion of annotations to the source code and (3) allow instrumentation of user-selected routines to support the dynamic analysis of the PUA.

The first requirement supports program browsing. The second one should allow the tool builder to insert *annotations* either into the code or (preferably) into the flow graphs of the selected routines. For example, following the approach in SPARK [1], one would want to insert **derives** annotations, essentially signatures of the mathematical functions computed by the subprograms in the PUA, to be used for the detection of possible data flow anomalies in the implementation. Also, one would like to be able to insert **assertions** in the language of the first level predicate logic, to be used by *program provers*, the ultimate form of static analysis.

One cannot overestimate the importance of the third requirement, the support of *Dynamic Analysis*. Typically, SAT tools are either/or affairs, supporting either static or dynamic analysis, with a possible exception of STAD [9],

which besides an extensive testing coverage also provides a modicum of static analysis (the detection of data flow anomalies, the identification of Definition-Use Chains and of Live Variables, cf [5]). We claim that *Dynamic Analysis should be an extension of Static Analysis*, rather than an activity in and of itself. Clearly, any code coverage testing tool needs a rudimentary static analysis to identify the parts of code whose coverage is to be monitored – statements, branches, decisions or data flow chains. More importantly, however, many important issues in static analysis are *undecidable*, i.e. there is no algorithm for their solution. For example, one might not know whether a particular branch in the program is feasible (executable), i.e. whether there is a program input that causes its activation. In principle, one could try to prove (or disprove) such a possibility. Unfortunately, the success of program proving depends on the programmer's ability to formulate appropriate assertions (hypotheses) and use the most promising inference rules. Those are not easy tasks. Here is then where Dynamic Analysis can be used.

To support Dynamic Analysis, the tool builder should be able to do at least the following. First, it should be easy to instrument user-selected subprograms and control points within them with calls to a (tool-builder) execution monitor. For example, one could define the following "active query"

```
procedure Instrument(subprogramId,  
subprogramId_instr, Monitor)
```

where `subprogramId_inst` is the name of instrumented (at the flow graph level) `subprogramId`, and `Monitor` is a procedure a call to which is inserted in front of every node in the flow graph, e.g. `Monitor(Node_number)`. Second, at each of such a call, `Monitor` should have a full access to all the variables of the program, thus being able to retrieve their values. Perhaps dynamic ASIS queries can be made available to directly use the object code for that purpose. Third, another "active" query should also be available to set breakpoints in the selected routine and allow a change in the PUA's state by redefining the values of selected variables before resuming the execution. Fourth, one should be able to dynamically evaluate the tool-user defined assertions (e.g. pre- and post-conditions, loop invariants).

The instrumented code then would produce several Dynamic Data Bases (DDBs) containing information about particular executions. Those DDBs can then be used by specialized execution-based tools such as test coverage monitors, debuggers etc.

7. REFERENCES

- [1] Barnes, J., "High Integrity Ada, The SPARK Approach," Addison-Wesley, 1997

- [2] Bergerreti,J.F., B.A.Carre, "Information-Flow and Data-Flow Analysis of while-Programs," ACM Trans. on Programming Languages and Systems, Vol.7, No.1, Jan 1985, pp. 37-61.
- [3] Ferrante, J., K.J.Ottenstein, J.D.Warrem, "The Program Dependence Graph and Its Use in Optimization," ACM Trans. Programming Language and Systems," Vol.9, No.3, July 1987, p.319-349, 1993
- [4] Graham, D.R., "Where is CAST Heading? Directions and Trends for Testing Tools," Proc. Sixth International Software Quality Week, San Francisco, May 25-28, 1993
- [5] Hecht, M.S., "Flow Analysis of Computer Programs," North-Holland 1977.
- [6] Steven. V.Hovater, Document Generation using ASIS Tools, Ada Letters, Dec 2000, p.40-49
- [7] Information Technology - Programming Languages – Ada Semantic Interface Specification (ASIS), ISO/IEC 15291: 1997(E)
- [8] Korel, B., J.Laski, "Dynamic Slicing of Computer Programs," Journal for Systems and Software, 1990, v.13, p.187-195.
- [9] Laski, J., "Data flow testing in STAD," The Journal of Systems Software,1990, Vol.12, pp. 3-14.
- [10]J.Laski, W.Stanley, J.Hurst, "Dependency Analysis of Ada Programs," Proc. ACM SIGAda Annual International Conference (SIGAda98), Nov. 8-12, 1998, Washington, DC, p. 263-275.
- [11]Laski, J.,W.Szermer, "Identification of Program Modifications and its Applications in Software Maintenance," *Proc. IEEE Conference on Software Maintenance*, Orlando, Florida, Nov. 1992, pp. 282-290.
- [12]Rossopoulus, N., Yeh, R.T., "SEES - A Software Testing Environment Support System," IEEE Trans. Softw. Eng., v.SE-11, No.4, April 1985, p.355-366.

8. APPENDIX

-- Example program. Comments in procedure Display
 -- identify nodes in the flow graph in Figure 7

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Float_Text_IO; use Ada.Float_Text_IO;
```

```
procedure Home_Budget is
```

```
Max_Expense : constant Float := 10_000.00;
type Categories is
  ( Food, Clothing, Housing );
```

```
package Category_IO is new
Ada.Text_IO.Enumeration_IO
  (Enum=>Categories );
subtype Expenses is
  Float range 0.00..Max_Expense;
type BudgetArray is array (Categories)
  of Expenses;

procedure Display (Budget: BudgetArray) is -- 1
begin -- 2
  For C in Categories loop -- 3,4,5
    Category_IO.put (c); -- 6
    Put ( ": " ); -- 7
    Put ( item => Budget(C), aft => 2,
      Exp=> 0 ); -- 8
    New_Line; -- 9
  end loop; -- 10, 11
  New_Line; -- 12
end Display; -- 13

procedure Get_Expense (
  Budget : out BudgetArray ) is
error : boolean;
begin
error := FALSE;
For C in Categories loop
begin
  New_Line;
  Put( "Please Enter " );
  Category_IO.Put ( C );
  Put ( " expense ... " );
  Get ( Item => Budget ( C ) );
exception
  when others =>
    Put("Sorry,Invalid amount! ");
  Ada.Text_IO.Skip_Line;
error := true;
end; --exception
If error then exit;
end if;
end loop;
new_Line(3);
end Get_Expense;

Budget : BudgetArray;

begin
  Budget := ( others => 0.00 );
  Display ( Budget );
  Get_Expense ( Budget );
  Display ( Budget );
end Home_Budget;
```