BEYOND ASIS: PROGRAM DATA BASES AND TOOL-ORIENTED QUERIES:

(Extended Abstract)

Janusz Laski, SofTools, Inc., Rochester Hills, Michigan 48307

email: laski@oakland.edu Internet: http://www.softools.org

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

1. INTRODUCTION. There have been suggestions around to define a second level of ASIS queries to Facilitate the development of Software Analysis and Testing (SAT) tools. In this paper we report on our experience with the development of an ASIS-based front end of SWAT (SoftWare Testing and Analysis), a comprehensive static and dynamic (execution-based) software analysis system that firmly confirms a need for such a level. Clearly, although ASIS indeed proved a great help, its learning curve was quite steep at times. This was particularly aggravating in the case of SWAT whose most important design objectives included: (1) integration of various SAT methods around a common conceptual framework; (2) an open architecture to allow the inclusion of new SAT methods and (3) the complementary use of static and dynamic (execution-based) analysis.

To achieve these objectives, the results of ASIS queries are stored in Program Data Bases (PDBs), to be used by specialized SAT tools. That approach was patterned on our earlier experience with STAD, a System for Testing and Debugging for Pascal [LASK90, LALU92] and inspired by [ROYE85]. Incidentally, it appears that the PDBs do provide enough information to build a variety of SAT tools; consequently they can be viewed as a second ASIS level. However, that is more due to the objectives of SWAT than to the idiosyncrasies of ASIS, essentially unknown to us at that time.

Here is the organization of the paper. In Section 2 the difficulties of using ASIS are illustrated. In Section 3 the (Primary) PDBs, both at system and procedural levels, are described. They are derived directly from the program source code. In Section 4 the derived PDBs are briefly described. Those are the product of processing the primary PDBs. Finally, in Section 5 future directions are discussed.

2. ASIS and SWAT. The Ada Semantic Interface Specification (ASIS, [IPAS97]) is a set of Ada packages that provides collections of queries about an Ada Program Under Analysis (PUA). Although ASIS proved a great help 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. 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 following simple statement:
IF A OR ELSE B THEN Integer_Variable := 1 + 2; END IF;

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).

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.

```
procedure Break_Into_Nodes(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.

3. PROGRAM DATA BASES. What follows is a series of brief descriptions of the contents of each Data Base produced by our Ada95 Front End. The Data Bases are actually ASCII text files
whose format was designed to be both readable by people and easily scanned by tool-builder programs. 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 Bases contain information about individual subprograms.

The following are SYSTEM DATA BASES derived by SWAT:

Elaboration Order Data Base (PUA.eor): lists all available compilable units in the program in the implementation-defined elaboration order.

InDeX (File-Unit) Data Base (PUA.idx): Provides a mapping between Ada units and the file in which the units are stored.

With Relations Data Base (PUA.wth): Identifies the With dependency in the program.

Routine Symbol Table Data Base (PUA.rst): 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. Cross-reference to the source is included.

Type Symbol Table Data Base (PUA.tst): Each type that appears in the program, whether standard or user-defined, is listed with its unique ID number. Anonymous types are also included.

Variable Symbol Table Data Base (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 Data Base, Type Symbol Table Data Base, and Index Data Base.

Routine Call Graph Data Base (PUA.rcg): For each subprogram R, subprograms potentially called by R are identified [HECH77].

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

The following PROCEDURE-LEVEL DATA BASES 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.

Control Flow Graph Data Base (package.cfg): 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 Data Base, they do not have separate flow graphs, as those are embedded in
the appropriate subprograms. Package specification (variable initialization) and body (variable initialization and initialization routine) have two distinct flow graphs.

Definition-Use Data Base (*.dfu): For each subprogram, variables used (referenced) and defined (assigned a value) in each node (instruction) in the subprogram are identified. A distinction is made between a total vs. partial definition of arrays, cf [LASH98]. 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): 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.

4. DERIVED DATA BASES AND HIGHER LEVEL QUERIES
The above data bases are derived directly from the source code and hence are viewed as "primary," since they contain the raw, further unprocessed information about the program. However, a SAT tool would most likely need to process several primary databases to get the information that it needs. The results of such processing are stored in Secondary Data Bases. Here are some examples:

Routine Sequence Graph (PUA.rsg), which provides a finer view of the calling structure of the subprograms than the PUA.rcg. Clearly, 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 [FEOW87]). The rsg allows one to identify all potential sequences of calls and, consequently, identify the invalid ones, such as Create(Stack); Pop(Stack).

An example of a procedural level derived database is the Definition-Use Chains (proc.duc) which identifies nodes in the flow graph that define a variable and those nodes in which the defined value can be used before being changed, if at all. Essentially, the .duc relation is equivalent to the first-level Data Dependency in the program.

Yet another derived artifact, albeit not really a database proper, is the instrumented code for a user-selected subprograms and control points within them to support dynamic analysis of the PUA. This involves inserting calls to a system monitor and providing the necessary information to print the values of the variables of interest together with the corresponding position in the execution trace. Alternatively, ASIS queries can be made available to directly use the object code for that purpose.

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.

5. FUTURE RESEARCH. No claim is made here that SWAT's PDBs offer an ultimate solution to the second-layer ASIS problem. Nevertheless, it seems rational to believe that the databases approach can indeed be used to build various SAT tools without using ASIS directly. However, further research into the power of the SAT methods is needed to identify an "optimal" set of second-level ASIS queries. It is natural to expect that the research will proceed in a piecemeal fashion, offering solutions to some subproblems before their conceptual integration takes place.
Clearly, at this point no particular subset of STA methods can be strongly recommended since (with the exception of formal proving and anomalies' detection) we don't yet know what we know. Indeed, all claims of 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. Having said that, it seems rational to start with the simplest, most promising and relatively easy to implement tools and test their usefulness. On the static analysis side, that would mean program dependencies of an arbitrary order [FEOW87, LASH98], the detection of anomalies [BARN97, BECA95] and system level views and documentation [HOVA2000]. On the dynamic analysis side, that would mean statement, branch, condition and data chain coverage [LASK90], profiles and dynamic (execution-based) data and control dependencies [KOLA90].

REFERENCES


