IDM -- A Configuration Management Tool for Maintaining Consistency Through Incremental Development

May 1, 1989

Technical Report #40

Robert L. Akers
Lawrence M. Smith

Computational Logic Inc.

1717 W. 6th St. Suite 290

Austin, Texas 78703

(512) 322-9951

This research was supported in part by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151., and by the National Computer Security Center, MDA904-87-C-6005. The views and conclusions contained in this document are those of the authors and should not be intepreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Defense Advanced Research Projects Agency, the National Computer Security Center, or the U.S. Government.

Abstract

This paper presents a mechanism which a software development environment may use to help maintain consistency among the various parts of a developing or evolving software system. Our view of the development process considers that 1) the program is represented as a collection of structured objects in a database and 2) the program development environment manages a collection of independent tools that operate on the database objects, storing their results in the same database. If one operation changes a database entry that is an input of another operation, then the result of the latter operation may be inconsistent with the database entries on which it should be based. We present a mechanism for managing the development process in such a way that these consistency problems can be resolved with minimal impact and show how it was used to implement a new Incremental Development Manager for the Gypsy Verification Environment.

1. Introduction

Change is a fact of life in software systems, particularly during development phases. The operations performed on one program unit by the program development environment (compilation of a procedure, for example) may rely on information about other units. Changing one of these other units can therefore introduce inconsistencies in the results of previous operations. A compiler does not extensively explore the network of interrelations among program units, and at worst, a system recompilation will resolve all inconsistencies between source code and compiled code and may cost only a few hours or days of computer time. Formal verification operations, such as verification condition generation and proof, tend to more extensively explore the interrelation network, however, increasing the number of operations which might be invalidated by a particular change in the program. Moreover, due to the high personnel cost of formal reasoning, redoing operations to restore consistency can be quite expensive. A program support environment with formal reasoning modules should strive to minimize the amount of work which must be redone to recover consistency after a source code change.

A program development support environment can be viewed as a collection of more or less independent tools coordinated to operate on a database. The database contains the program under development and data objects associated with it, e.g. proofs, cross-references, object and executable files, etc. We say a *USE* of a data object by an operation occurs when the data is utilized in the operation. With this view, we can discuss the problem of maintaining consistency through incremental development in terms of the general problem of maintaining consistency of a changing database with a network of internal dependencies. The dependency network is represented by the collection of all USES which the environment has recorded.

This paper presents a paradigm, suitable to this view, for managing incremental development. The paradigm allows dependencies to be tracked to an arbitrary level of granularity, which helps to minimize the amount of work to be redone in response to a change. The paradigm is more closely tied to the relationship between the program database and the environment tools than to any notion of programming language semantics. Therefore, it is applicable to a wide variety of program development operations. (In fact, it is applicable to database consistency problems not associated at all with program development environments.)

The paradigm was recently used to implement a fine-grained Incremental Development Manager (IDM) for the Gypsy Verification Environment (GVE). [Good 88] Gypsy is a collection of methods, languages, and tools for building formally verified computing systems. [Good 89] The GVE provides capabilities for specifying a system, implementing it, and for using formal, logical deduction to prove properties about its specification and implementation. The most fundamental operations are presenting a program for parsing and type checking, generating verification conditions (VCs), which are conjectures whose proofs validate the consistency of the program with its specification, and proving these VCs with the help of an interactive prover. The IDM is responsible for helping to maintain consistency between the results of these operations and their inputs. The new IDM replaces an older tracking facility which was known to be flawed.

The rest of this paper will summarize previous work, outline the implementation of the new GVE mechanism, and reflect on these results.

2. Background

An excellent paper by Perry [Perry 87] presents a taxonomy of software interconnection models, discussing examples of each. Perry's principal categories are those based on: 1) units (typically files), 2) syntactic program objects (procedures, types, etc.), and 3) formulas capturing semantic information about program objects. Incremental development systems can be discussed quite naturally in terms of software interconnection models.

The most common means of tracking incremental development is based on dependencies among files. Among the many systems that use files and file versions as the unit of development granularity are the Unix Make [Feldman 86], the Software Change Control System (SCCS) [Rochkind 75], and the Revision Control System [Tichy 85]. None of these systems consider the semantic content of the files they manage; neither do they deal with granularity at or below the level of a program unit, such as a procedure. Therefore they can be described in terms of Perry's category of unit interconnection models.

A step from unit interconnection to syntactic interconnection was made by Tichy [Tichy 86] in a tool that determines when changes to a program entity necessitate recompilation of various parts of a system.

More than any of the aforementioned systems, the Boyer/Moore theorem prover [Boyer & Moore 88] solves a problem akin to that faced in the GVE: proof dependencies that penetrate arbitrarily deeply through the program structure. The prover enforces a rigid incremental development policy for function definitions. Any time a function is redefined, all definitions and proofs which have ever depended in any way, either directly or transitively, upon the function being redefined are immediately discarded. While this Spartan approach guarantees soundness and consistency, it does not offer much flexibility to the user, nor does it attempt to preserve previous work which depends on unmodified parts of a redefined function.

The INSCAPE Environment [Perry 89] uses semantic interconnection information to track dependencies among program modules, with a goal of supporting formal modularization of large scale systems. The environment tracks pre- and post-condition dependence, caller obligations, and preclusion and handling of runtime exceptions. User-defined predicates serve as the points of interconnection. The INSCAPE incremental development tool, Infuse [Kaiser 87], suggests when a change in the interface or implementation of a module might require changes to other interfaces or module implementations. This is a related but somewhat different problem than that addressed by the Boyer-Moore mechanism and the GVE, which are largely concerned with the consistency of a collection of proofs derived from code.

The incremental development algorithm employed in the GVE in years past [Moriconi 77] [Moriconi 79] is known to have been deficient in both concept and implementation. Among its principal shortcomings were that:

- It was not based on a general model independent of the operations involved. Rather, an internal model of GVE operations and relationships was an intrinsic part of the design and implementation of the tool. The incremental development "rules" were deeply embedded in both the theory and the software implementation. This discouraged a clean factoring of concerns.

- It noticed only in a coarse way the connections between program units, specifically noting only that some information from a particular unit had been used without noting what that information was.
- To compensate for the lack of detail, it made assumptions about the nature of the USE. For example, it assumed that if a function was USED in parsing, only its header was utilized, or if it was USED in VC generation, only its external specifications were touched. The rules assumed these characterizations of the USED relationships among units, and utilized these assumptions to declare what operations would be called into question when a program unit was changed. This led to an idealized theory which seemed reasonable, but which had no direct connection to the real interrelations established during program development.
- Syntactic references in the Gypsy code, rather than operational references to the objects in the GVE database, were sometimes used for determining the presence of USES. This is generally inadequate, since USES in some operations may be transitive down the reference tree to varying degrees, depending on the implementation of the program development environment.
- The original design was incorrect, even in its own ideal terms, with respect to dependencies arising during proof. This design was never implemented to support the GVE prover.

3. The Implementation of the IDM

Although our incremental development paradigm is quite generally applicable, it will be helpful to describe it in terms of its implementation as the new GVE Incremental Development Manager.

Some definitions will help make the discussion clear.

1. A Gypsy program *UNIT* is a procedure, function, lemma, constant, or type.
2. A *SYNTACTIC COMPONENT* of a Gypsy unit is a separately identifiable portion of a unit. Examples are a function’s parameter list, the exit specification of a procedure, and the value of a constant.
3. A GVE database *OBJECT* is the internal representation of a unit.
4. An *OPERATION* is an analysis the GVE does on an object. The operations covered by the IDM are semantic checking of a unit (a phase of parsing), verification condition (VC) generation of a unit, and proving a verification condition. Other operations which could be tracked are compilation, information flow analysis, or documentation assemblage; in short, any action that uses database input.
5. A *USE* is a dependency, by one object A on another object B, created by a particular operation on A. If B changes because of incremental development, then that operation on A must be checked or possibly redone.

The key steps we take in managing incremental development are first to note when an operation uses certain input data, and then to recognize that when that data is changed, the operation must be checked. The observations the incremental development manager makes about the USE relations among program units can be as fine-grained as the property structure of the underlying database. The tool makes absolutely no assumptions about how these relations manifest themselves in various development operations.

The overview of the IDM is straightforward.

1. The USES associated with each operation are recorded as information is fetched from the database.

2. The USE information is inverted, so that each unit has not only a list of all USES stemming from operations based on the unit, but also a list of inverted USES showing which operations on other units USED this one.
3. When a unit changes, the inverted lists allow the relevant operations on units to be efficiently marked for checking.
4. On the basis of program status reports, the user issues commands to check the marked operations when he sees fit.

The IDM implements a general scheme for incremental development. A GVE-specific rule base provides direction for actions which maintain database consistency. The rule base is driven by a propagating database manager, which triggers these actions when new information is stored in the database.

In describing the mechanism, we will first sketch the structure of the GVE database, then describe its new rule-based propagating database manager. With that background, we will discuss the recording of USES of program objects. Then, we will exhibit a critical propagation rule and show how it completes the information gathering phase. Finally, we will describe how incremental development status and cross reference information is reported to the user.

3.1 Our Example

We will use a simple Gypsy example to illustrate the operation of the mechanism. Knowledge of the Gypsy language and of the GVE is helpful but not required for understanding.

We have a recursive functional specification for the mathematical function **factorial** and a procedure **fact** that computes it iteratively. The goal of our verification is to prove that the execution of **fact** computes **factorial**. The proof, however, is not the point of the example. Rather, we will investigate how incremental changes to the program affect the work that has already been done.

Here is a fragment of Gypsy source code for the example.

```
scope factor= begin

  const maxpos :integer := 32767;
  type small_int = integer [1 .. maxpos];

  procedure fact (n:small_int; var fact_result : small_int) =
    begin
      exit fact_result = factorial(n);  { specification of fact }
      { .. the executable body of fact .. }
    end  {fact};

  function factorial (m:small_int): small_int =
    begin exit (assume factorial(m) =
               if m = 1 then 1 else (m * factorial(m-1)) fi);
    end  {factorial};
end;  {scope factor}
```

3.2 The GVE Database Structure

To understand the mechanism, it helps to understand the GVE database structure.

There is one database object for each Gypsy program unit. Each database object has a set of attributes which, in turn, may have a number of entries. Among the attributes of a database object are the unit's semantically analyzed internal representation (which we call its INTERNAL-REP), its verification conditions, a structure describing its verification status, and a structure containing cross reference relationships with other units. We will discuss each of these principal attributes in turn.

The INTERNAL-REP is constructed by the parser during semantic analysis of the unit. It contains, among other things, an internal representation of each syntactic component of the unit. Consider our Gypsy routine **fact**. The INTERNAL-REP contains separate entries for the formal parameter list (header), the exit specification, the local variable declarations, and the executable body. There is also an I-USE list entry, wherein every USE of another unit during semantic analysis is noted.

The status attribute of a unit has a set of entries describing such things as whether its semantics need checking because of an incremental change to a unit USED in the original semantic analysis, whether the unit has had its VCs generated, and whether its VCs or proofs need to be checked because of a change to a unit USED during VC generation.

Among the entries in the verification condition attribute are a list of VCs for the unit and an I-USE list corresponding to USES occurring during VC generation. Each of the VCs contains a representation of its proof (if it has been attempted), an indicator of the status of the proof (e.g., trivially true, proved, partially proved, or not attempted), and an I-USE list corresponding to USES occurring during the proof.

The cross reference attribute contains an entry for each syntactic component of the unit. The values under that entry indicate which other units in the database have USED that component. We call these entries THEY-REF properties. In our example, the unit **factorial** will have a THEY-REF-MY-HEADER entry which notes that the semantic checking of **fact** USED the header of **factorial**. Note that the I-USE lists for a unit **foo** indicate which units (and which parts of those units) were USED in the analysis of **foo**, while the various THEY-REF lists of **foo** indicate which units USED the various parts of **foo**. We maintain the invariant that, collectively, the combined THEY-REF properties are an exact inversion of the combined I-USE lists for all units in the database.

3.3 The Propagating Database Manager

The objects of the GVE database and their attributes are managed by a rule-based propagating database manager [LSmith 88]. The database designer can associate rules with an arbitrary database attribute, attribute entry, or, generically, with any node in the database structure. The rules are triggered when the value of the attribute (or node) to which it is attached is modified. The role of the rules is to describe actions which can keep the database in a state that is consistent with respect to the changed value.[1]

A rule has three components:

1. A predicate, **Pred**, which must be satisfied in order for the propagation to occur. The evaluation of this predicate cannot cause a database attribute to be stored.
2. A **Form** which is evaluated to effect the propagation.
3. A collection of attributes, **Attr-set**, which are allowed to be modified as part of the immediate propagation.

When the value of an attribute with a propagation rule is stored, if the new value is different than the old one, the database manager invokes the propagation rule and takes the following action. First, **Pred** is evaluated. If **Pred** is satisfied, the propagation **Form** is evaluated. **Form** can do anything it wishes, except that if it directly stores a value in the database, the target attribute must be among the ones given in **Attr-set**. The attributes whose names are in **Attr-set** may also have propagations attached to them, so that

[1] In fact, the rules can cause arbitrary actions, but we will discuss only their role in incremental development.

a chain of propagations may be triggered by storing an attribute value in the databsae.

Propagation rules are used to build and maintain both the status attribute structure mentioned above and the collection of cross reference THEY-REF properties.

3.4 Marking the USES of Program Objects

Our implementation employs the notion of *active I-USE lists*. For every type of I-USE list (I-USE-IN-PARSE, I-USE-IN-VCGEN, I-USE-IN-PROOF, etc.), there exists a single I-USE list variable in the GVE implementation.[2] When a context is entered, for instance when the parser begins checking a particular unit or when a proof of a VC is begun, the appropriate I-USE list is *activated* and is initially empty. Then, whenever information from another unit is USED, (for instance when there is a database query to fetch the type definition of **small_int**), we record on all active I-USE lists that the type definition of **small_int** has been USED. This recording is done in conjunction with the database fetch. When we exit the context for the active I-USE list, the list is deactivated, and the value of the I-USE list variable is stored in the database. I-USE lists may be reactivated with initial values from previous computations.

From a software engineering viewpoint, the advantage of this recording mechanism is a proper separation of concern. I-USE lists can be activated at a high level of the implementation, where context is typically controlled, without regard for what kind of lower level activity might trigger a USE or what the USES might be. But the actual recording of a USE will normally occur at the low-level of the implementation, where the database is accessed, without concern for the context of the larger computation. This factoring of duties makes arbitrarily fine granularity possible while contributing to the modularity and overall reliability of the system.

We mentioned that the old GVE IDM made gross assumptions about which information is USED in the GVE. The new IDM avoids this possibility completely by recording USES precisely at their point of occurrence. There is no question of interpretation, and unlike the old system, there are no inferences based on mere expectations about how certain operations are conducted and how incremental development might affect them. Everything is strictly realized in the implementation.

Thus far, we have not discussed the granularity of information recorded in an I-USE list, except to say that it was coarse in the previous implementation of the IDM. In fact, all that was noted in the old IDM was the name of the unit which was USED. Moreover, there was only one I-USE list for each unit, and it was utilized for all operations. This lack of granularity extended into the I-USE list cross references.

The granularity is much finer in the new IDM, and this is one of the secrets to its success and its reliability. Each entry under an I-USE list is of the form:

```
entry ::= ( <unitname> ( <use tag>+ ))
```

where the tags indicate what parts of the unit were USED. The part of an I-USE-IN-PARSE list derived from the exit specification of **fact** might be:

```
(.. (factorial (ITS-KIND ITS-HEADER))
    (small_int (ITS-TYPE-DEF)) ..)
```

---

[2]This implies an assumption that only one unit is being analyzed at a time. This is not strictly necessary. In general, it is only necessary that there be some capability for associating some I-USE list variable with every operation/context pair. Moreover, the same operation can be associated with multiple operations or multiple contexts simultaneously. For instance, we have simultaneously active I-USE lists for the parsing of a unit and for the parsing of a particular syntactic component of the unit, e.g., its exit specification.

This means that in the semantic analysis of **fact**, the parser USED the KIND of **factorial** (to note that it is a function) and USED its formal parameter list (to make sure the actual parameters are compatible with their formals), and it also USED the type definition of the formal, **small_int**, (to make sure it matched with the type of the actual, also **small_int**).

The granularity of the USE tags carries into the inversion of the I-USE lists into THEY-REF lists. In this case, the unit **small_int** will have a THEY-REF-MY-TYPE-DEF entry among its THEY-REF attributes. A partial picture of that entry will be:

```
(THEY-REF-MY-TYPE-DEF .. (fact (IN-ITS-PARSE)) ..)
```

This means that during the semantic analysis of unit **fact**, we USED the type definition of **small_int**. To extend this example, if the type definition of **small_int** were examined during the VC generation of another unit **gcd** and during the proof of a VC **fact#3**, the above entry would look like:

```
(THEY-REF-MY-TYPE-DEF .. (fact (IN-ITS-PARSE))
                         (gcd (IN-ITS-VCS))
                         (fact#3 (IN-ITS-PROOF)) .. )
```

As we shall see, the THEY-REF properties are the linchpin of the system's response to an incremental change to a unit.

3.5 Two Critical Propagation Rules

Recall that each operation is responsible for collecting the USE dependencies that it creates, and these cross-references are completely inverted in the database by the propagating database manager. When, for example, the VC generation of unit **fact** USED the entry spec of **factorial**, that information was recorded as a USE in an I-USE list property of **fact** and its inversion stored in a THEY-REF property of **factorial**.

The propagation rule attached to all I-USE list attributes triggers this inversion. We state it informally, where the I-USE list belongs to a unit **A**:

```
PRED: true

FORM: First determine which USES in the old value are absent in the new one
      (call these DELETES) and which USES in the new value are not present in
      the old value (call these ADDS).  DELETES and ADDS are of the general
      form of a USED list.  Now invert the DELETES and ADDS, for example
      turning A's USED list entry:

          (I-USE-IN-VCGEN (B ITS-HEADER) (C ITS-TYPE-DEF))

      into a candidate for B's THEY-REF entry

          (THEY-REF-MY-HEADER (A ITS-VCGEN))

      and C's entry

          (THEY-REF-MY-TYPE-DEF (A ITS-VCGEN)).

      Finally, remove the inverted DELETES from the appropriate THEY-REF lists,
      then add the inverted ADDS to the appropriate THEY-REF lists.

ATTR-SET: (THEY-REF-MY), i.e., the THEY-REF structure (which encompasses
          all the THEY-REF lists) of any unit may be modified.
```

Another critical database propagation rule is attached to each syntactic component of INTERNAL-REP. It is informally stated as follows:

```
PRED: The syntactic component changes in a significant way,[3]

FORM: Get the THEY-REF list for that syntactic component of the unit, and
      for each operation mentioned in a use tag, mark that operation as
      needing checking for the unit to which it applies.

ATTR-SET: The status attributes which indicate that the semantics of a unit
          need checking, that its VCs need checking, and that any individual
          VC needs its proof checked.
```

Note that the propagation form says only that the relevant operation should be marked as needing checking. It does not actually perform the checking. Since the checking is potentially expensive, we let the user do it when he sees fit. He may wish to make several successive changes which would each affect one operation, for example, and this allows him to defer checking the operation until after he has made all his changes, if he so chooses.

We can see this rule at work if we change the exit specification of **factorial** to be:

```
    exit (assume factorial(m) =
               if m lt 2 then 1 else (m * factorial(m-1)) fi);
```

We shall see below how this change could cause a change in the verification status of **fact**.

3.6 Reporting Program Status, Cross Reference, and Enforcement

Propagation rules are used to build and maintain both the status attribute structure mentioned above and the collection of cross reference THEY-REF properties. These two sets of properties are used by the IDM to maintain an efficient representation of the development status of the entire program. From these properties and the I-USE lists, detailed status reports and cross references can be delivered to the user.

For this status report, the example has been parsed, the VCs generated for **fact**, and the VCs **fact#2** and **fact#3** proven interactively.

```
Gve -> show status
       Units and/or VCs -> fact

   VCs generated, not all proved:
         FACT

   FACT
      Correctness vcs:
         Basic vcs, proved in theorem prover:
           FACT#2, FACT#3
         Basic vcs, proved in VC generator:
           FACT#1
         Basic vcs, not yet attempted:
           FACT#4, FACT#5
         Well-formedness vcs, proved in VC generator:
           FACT#1
```

The cross reference report at this point would be:

```
Gve -> show crossref
       Units and/or VCs -> fact
FACT (a Procedure) refers in its parse to:
        FACTORIAL: its header, its kind.
        MAXPOS: its value, its type specification, its kind.
```

[3] At present, any change is significant, but we could loosen this restriction in some contexts, for instance if we are simply renaming a variable throughout a unit, or if we are relaxing a specification constraint in such a way that all existing proofs will still be valid.

```
        SMALL_INT: its type definition, its kind.

FACT refers in its VC Generation to:
        FACTORIAL: its entry spec, its exit domain spec, its header.
        MAXPOS: its type specification, its kind.
        SMALL_INT: its type definition.

   References to FACT:
        *no units*

VCS for FACT:
FACT#2 (a vc) refers in its proof to:
        MAXPOS: its value, its type specification, its kind.
        SMALL_INT: its type definition.

FACT#3 (a vc) refers in its proof to:
        FACTORIAL: its entry spec, its header, its exit spec, its kind.
        MAXPOS: its value, its type specification, its kind.
        SMALL_INT: its type definition.
```

After making the modification to the exit specification of factorial, the status report is:

```
Gve -> show status
       Units and/or VCs -> fact

   VCs generated, not all proved:
         FACT

   FACT
      Correctness vcs:
         Basic vcs, proved in theorem prover:
           FACT#2, FACT#3
         Basic vcs, proved in VC generator:
           FACT#1
         Basic vcs, not yet attempted:
           FACT#4, FACT#5

         Basic vcs, incremental development has marked the proof to be checked:
           FACT#3
         Well-formedness vcs, proved in VC generator:
           FACT#1
```

Note that since neither the semantic analysis nor the VC generation of **fact** referred to the exit specification of **factorial**, neither of those operations were marked to be checked. In the proof of **fact#3**, however, we USED **factorial**'s exit specification, and therefore the proof needs to be checked.

The status attribute and cross reference information is also used to inhibit operations which are likely to immediately become inconsistent. For example, we inhibit the VCGen operation on units which need semantic checking or units whose semantic checking USED other units which now need semantic checking. Similarly, we will not allow the user to enter the prover with a verification condition which USES units which need semantic checking, or a verification condition of a unit which is marked to have its VC's checked.

4. Discussion and Extensions

4.1 Payoff

The GVE is a large and complex software development environment, and the wrong model of incremental development could result in an unmanageable set of incremental development rules. When we first developed our propagating database manager, we envisioned a set of rules in the style of Moriconi but extended to cover all development scenarios. We imagined the need for hundreds of rules and dreaded the task of guaranteeing their consistency. After completing our model of the database and formulating the algorithm for the fine-grained marking of USES, we recognized there were only *two* propagations of central importance: the inversion of I-USE lists to THEY-REF lists, and the marking rule illustrated in the example above! After tying up all the details, we found only ten distinct propagation rules necessary to completely capture incremental development in the GVE. The design of propagation driver, the database, and the marking algorithm had paid off in a big way.

4.2 The Complete Set of Propagation Rules

The complete set of distinct propagation rules are as follows:

- Invert I-USE lists into THEY-REF lists, as described in 3.5.
- Mark operations which need checking, as described in 3.5.
- When a unit is parsed with semantic errors, note that its semantics will need to be checked.
- When a unit with errors is re-parsed and the errors corrected, turn off the flag which says its semantics need checking.
- When a new unit is parsed, initialize its VCG-STATUS as either NOT-GENERATED or NOT-POSSIBLE (i.e., with a type or a constant).
- When a VC is generated, record its status as either being TRUE, FALSE, or PROOF-NOT-ATTEMPTED.
- When a unit with VCs is changed, turn its CHECK-VCS flag on.[4]
- When a proof is attached to a VC, set the VC-STATUS to PROVED if the proof is complete, or to PROOF-SUSPENDED if the proof is partial.
- The composition access lists of abstract types are inverted, so that each unit has a property which shows the abstract types to which it has access.
- When a type is deleted, update an internal structure which contains information about all declared types.

(The last two items have no direct bearing on incremental development or status, but are of general utility to the GVE. There is no limitation on what propagations can do, beyond the constraints of the allowable attributes list in the propagation.)

4.3 Unexpected USES and Optimization

After installing the rules, we noted the phenomenon that the implementation was USE-ing more information than one might naively expect. These instances seemed to break down into four classes, each of which required a different GVE implementation response.

1. Unanticipated but correct USES-- For instance, when expanding a function during proof, the system extracted information from the entire call tree of the exit specification of the expanded function. This was because the GVE does not support proof of mutually recursive functions, and call tree traversal was necessary to determine if mutual recursion was present.

---

[4]This would be subsumed by the second rule if a unit's USES of itself were recorded.

Naturally, we uphold these USES.

2. Spurious USES -- There were cases where the implementation performed computations unrelated to the decisions at hand, for instance by utilizing service functions which were designed for other purposes but which fortuitously returned the desired results. The remedy was to tune the implementation to use more appropriate utilities, thus improving efficiency.

3. Non-optimal algorithms -- We uncovered a number of cases where information was in fact being legitimately extracted and its USE marked, but the computation was more complicated than it needed to be, perhaps because of unnecessary generality. The remedy was to optimize the algorithms, resulting in a reduced set of USES and, again, improved efficiency.

4. Non-impact USES -- This is the most subtle case. Occasionally, a computation USES information in such a way that incremental development of the USED units would not change the result of the computation or otherwise require any checking. For example, in the algebraic simplifier, a performance optimization gets information about the symbols in an expression, causing USES of types. This allows the simplifier to return a result much more quickly, but the same result would have been returned without the optimization and the USES. There is no reason, then, for the USES to affect incremental development. In cases like this, we can temporarily disable or otherwise restrict the recording of USES. Decisions such as this, to bypass the basic incremental development mechanism, should be made with great care.

It was natural to take a very conservative approach in our initial implementation of the new incremental development paradigm. Optimizing the implementation is an open-ended task.

In retrospect, we feel that the very introduction of the IDM provided a measure of inter-connectivity in the GVE implementation we did not previously have. With this knowledge, we were able to improve the quality of the implementation in areas not directly related to incremental methods.

4.4 A Note on Caching

In the GVE, as in many systems, caching information is an important strategy for increasing efficiency. In implementing an IDM in this style, it is critical to recognize that when an operation caches information, the USES that occurred while computing that information must also be cached.[5] When the cached information is utilized, the cached USES associated with the information must be recorded in the current context. This complicates any caching mechanism, but unless it is done correctly, there will be a hole in the tracking of dependencies.

4.5 Extensions to the Work

Aside from the open-ended optimizations mentioned earlier, there are several directions for future IDM-supported work.

The granularity of contexts associated with I-USE lists in the GVE is not as fine as it could be. Although the parser stores an I-USE list for each syntactic component of a unit, the VC generator stores only a single I-USE list for all the VCs for a unit, encompassing the entire VC generation operation for that unit, and the prover stores only a single I-USE list for each proof. In general, the smaller the contexts of the operations, or the finer the subdivisions of an operation, the more granularity can be realized in I-USE lists and in incremental development tracking.

In an entirely different vein, the incremental development information currently stored in the database

---

[5] Note that this discussion applies to operations caching information, as opposed to the database system. If we picture the IDM as a layer sitting between the database system and the various system operations, we are concerned with caching above the IDM layer.

could support a knowledge-based tool for guiding the user through the verification task. This could take a number of forms, one of which might be a modification impact analysis tool, or a "What if.." facility.[6] For instance, a user could ask what would be the effect of changing the exit specification of a unit, and he might be told that he would have to re-parse the unit, to regenerate VCs for that unit and perhaps some others, and to replay or re-do certain proofs. We might consider extending this report with a "why is that?" annotation, which would describe what activity (e.g., expanding a function during a proof) caused the dependency to exist. Note that with the separation of concerns in our implementation, this extension would be messy, since the reason for the USE, which is probably known at a high level of the implementation, would have to be communicated to the lower-level functions which recorded the USE.

We envision that the entire IDM could be cast as a stand-alone tool which could be layered over any object-oriented database system and then be used by any set of tools which could be arranged to use the appropriate IDM interfaces. A paper is being drafted which describes the formal model of incremental development embodied in the IDM and which could be used as the benchmark for evaluating the stand-alone tool. We would hope that the tool could be coded in a language suitable for formal verification and that its desired properties could be verified by proof.

4.6 Applicability

This discussion has concerned itself only with the way the IDM paradigm is employed to achieve high reliability configuration management of Gypsy programs and specifications. We feel, however, that the paradigm is applicable to a much broader set of problems.

For example, consider the task of releasing new versions of a software development environment to users who have work in progress on the platform of the old version. Perhaps a change in the new release might affect the work in progress. In the GVE, for instance, a change to the VC generator might be significant enough to warrant checking of VCs under certain circumstances. The IDM could be employed to "stamp" the unit with the version of the VC generator used to create its VCs. If a subsequent release called for checking of VCs, the database object representing the VC generator would be given a new value for its version attribute. When the modified value was stored, a propagation rule would note which units had VCs associated with the old VC generator and would mark those units as needing their VCs checked with the new VC generator.

The IDM would lend itself to the sort of straightforward configuration management measures being implemented in various program support environments. The notion of "created after" relations on files, which are the basis of incremental development tracking in many systems, could be captured with timestamps and coarse USE recording for files. This rather crude use of the IDM could be significantly refined for any identifiable components of files that could be represented in a database.

Of course, for languages that support the incremental compilation of units, the IDM would support the simple problem of knowing what program units throughout a system need to be incrementally recompiled after changes have been made in particular units.

It seems that the model described here could be applied to the fully general problem of maintaining consistency of an arbitrary database with various data dependencies when any database element is updated. The principal problem would be optimizing the space requirements for storing the incremental development properties, with the tradeoff being storage vs. granularity.

---

[6]Moriconi provided an ad hoc version of such a tool in his earlier work.

5. Conclusion

We have presented a mechanism for managing incremental development and illustrated its support of the development of verified Gypsy programs. To achieve high reliability, the model we use interprets incremental development events strictly in terms of the implementation of the programming environment; a USE is recorded because something was fetched from the database, not because some model of the verification process stated that a USE was likely. Our implementation employs a separation of concerns which strongly reinforces the reliability of the system. Its observations have a fine degree of granularity, which tends to minimize the job of restoring consistency to a verification and proof state in which some units have been changed. The IDM itself became a useful software analysis tool for our program development system by helping to reveal the subtle inter-connectedness in its implementation.

While this discussion focused on a particular software development methodology and its tools, we see no reason why it could not be applied to a broad class of problems in program development environments and, in general, to manage update propagation in any database system.

References

[Boyer & Moore 88]
R. S. Boyer and J S. Moore.
A Computational Logic Handbook.
Academic Press, Boston, 1988.

[Feldman 86] S.I. Feldman.
Make -- A Program for Maintaining Computer Programs
University of California, 1986.
Unix Programmer's Manual Supplementary Documents I.

[Good 88] Michael K. Smith, Donald I. Good, Benedetto L. DiVito.
Using the Gypsy Methodology
Computational Logic Inc., 1988.
Revised January 1988.

[Good 89] Donald I. Good, Robert L. Akers, Lawrence M. Smith.
Report on Gypsy 2.05
Computational Logic Inc., 1989.
Revised January 10, 1989.

[Kaiser 87] Dewayne E. Perry and Gail E. Kaiser.
Infuse: A Tool for Automatically Managing and Coordinating Source Changes in Large Systems.
In *Proceedings of the 1987 ACM Computer Science Conference*. ACM, St. Louis, Mo., February, 1987.

[LSmith 88] Larry Smith.
The Propagating Database Manager.
Internal Note 78, Computational Logic, Inc., September, 1988.

[Moriconi 77] Mark S. Moriconi.
A System for Incrementally Designing & Verifying Programs.
PhD thesis, The University of Texas at Austin, 1977.
ICSCA-CMP-9.

[Moriconi 79] Mark S. Moriconi.
A Designer/Verifier's Assistant.
IEEE Transactions in Software Engineering SE-5:4:387-401, July, 1979.

[Perry 87] Dewayne E. Perry.
Software Interconnection Models.
In *Proceedings of the Ninth International Conference on Software Engineering*. IEEE Computer Society, March, 1987.

[Perry 89] Dewayne E. Perry.
The Inscape Environment.
Technical Report, AT&T Bell Laboratories, February, 1989.
publication pending.

[Rochkind 75] M.J. Rochkind.
The Source Code Control System.
IEEE TSE SE-1:364-370, December, 1975.

[Tichy 85] Walter F. Tichy.
RCS -- A System for Version Control.
Software -- Practice and Experience 15:7:637-654, July, 1985.

[Tichy 86] Walter F. Tichy.
Smart Recompilation.
ACM Transactions on Programming Languages and Systems 8:3, July, 1986.

Table of Contents