Fine Grained Data Management To Achieve Evolution Resilience in a Software Development Environment

Richard Snodgrass
Department of Computer Science
University of Arizona
rts@cs.arizona.edu

Karen Shannon
Department of Computer Science
University of North Carolina
shannon@cs.unc.edu

Abstract

A software development environment (SDE) exhibits evolution resilience if changes to the SDE do not adversely affect its functionality nor performance, and also do not introduce delays in returning the SDE to an operational state after a change. Evolution resilience is especially difficult to achieve when manipulating fine grained data, which must be tightly bound to the language in which the SDE is implemented to achieve adequate performance. We examine a spectrum of approaches to tool integration that range from high SDE-development-time efficiency to high SDE-execution-time efficiency. We then present a meta-environment, a specific SDE tailored to the development of target SDE's, that supports easy movement of individual tools along this spectrum.

A software development environment (SDE) is a collection of tightly coupled tools cooperating to facilitate the activities of design, implementation, testing and management involved in producing a software artifact. An SDE is itself a large collection of tools, comprising 100K to 1M lines of code and involving significant development effort by a team of programmers over several years. The close interaction of tools required to achieve integration unfortunately complicates their implementation, as changes to a tool or to the structure of the shared data will necessitate changes to other tools. Due to the size and complexity of the SDE being developed, evolution is a constant occurrence; the challenge is in reducing its costs, in terms of programmer time and effort and execution efficiency [Taylor et al. 1988].

An SDE exhibits *evolution resilience* if it meets two requirements. First, changes to the SDE must not adversely affect its functionality nor performance, to ensure that the end product performs correctly and efficiently. Second, there must not be significant delays in returning the SDE to an operational state after a change, to ensure that the development effort required to realize the end product is minimized. Implementation techniques, development strategies, and support software that increase the evolution resilience of the emerging SDE are needed [Wileden et al. 1990].

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1990 ACM 0-89791-418-X/90/0012-0144...$1.50

In this paper we discuss the kinds of evolution that occur during development of an SDE, and the often substantial impact of these changes. We focus on fine grained data, in part because the performance requirements concerning fine grained data are the most difficult to meet. A spectrum of approaches to tool interaction is presented; this spectrum incorporates and generalizes several approaches previously proposed by others. We examine implementation techniques that allow tools to be moved individually along this spectrum with relative ease, significantly ameliorating the adverse impact of evolution. Finally, we discuss specific facilities that when present allow the SDE developer to precisely control the evolution of the emerging SDE, and briefly examine our implementation of these facilities.

1 The Problem

As discussed above, the defining characteristic of a software development *environment*, as opposed to simply a collection of tools, is the close interaction of the tools enabled by their sharing of information concerning the program under development [Barstow et al. 1984]. An SDE is employed by an *environment user* (often a team) to develop a *software system* consisting of the software itself, along with supporting specification and design documents, test data, management reports such as schedules, and other artifacts necessary for continuing maintenance and enhancement of the software system. We are concerned here with an earlier activity: the development of the SDE itself by an *environment developer* (almost definitely a team). SDE's also evolve over time, initially as they are implemented but also as they are maintained and extended by the developer. Tools in the SDE must be easy to change, it must be possible to generate an executable version of the modified portions of the SDE in a timely fashion, so that the environment developer's time is not wasted, and the resulting SDE must be highly efficient if it is to be employed by the user to develop large software systems.

1.1 Characterizing Evolution of an SDE

There are four primary ways that an SDE may evolve. *Data specification evolution* refers to changes in the kinds of information stored in the central database. Examples include adding a new object type such as a bug report, changing

the name of an attribute in a symbol table entry, or moving an attribute from the database into an individual tool, if that attribute contains information relevant only to that tool and to a single execution of the tool. *Representation evolution* refers to changes in the specific encodings of the information, both in the database and within the tools. Examples include increasing the precision of integer constants stored in a symbol table, changing the implementation of a sequence of objects from a linked list to pointers embedded in the objects themselves, and reordering attributes within an object. *Tool evolution* involves changes to the algorithm embodied in a tool; examples include changing the way outstanding bugs are assigned to programmers, or modifying an equation in an attribute grammar specification for semantic analysis. Finally, *system evolution* deals with changes made at an architectural level. Examples include adding a cross reference tool, or augmenting a low level debugger to support the display of information in terms of the source code; both may involve fairly significant changes to the central database and to several tools in the SDE [Wileden et al. 1990]

Each kind of evolution impacts the developing SDE. It can affect the semantics of existing data: new attributes or objects may have to be computed, and other attributes or objects may have to be deleted or changed. Additionally, the representation of existing data stored in the database may have to be modified. Finally, the representation of data inside tools may have changed, necessitating at worst modification of the code by the environment developer and generally at least a recompilation of the tool [Conradi & Wanvik 1985].

This paper focuses on approaches to tool integration that support all four kinds of evolution, while minimizing the impact in its various guises of that evolution. Our concern primarily is the data management support necessary to accommodate evolution of an SDE, while preserving SDE-execution-time efficiency. We assume that smart(er) recompilation [Schwanke & Kaiser 1988, Tichy 1986] is employed to eliminate redundant computation. We do not consider here techniques such as parser generators, attribute grammar systems, and other approaches that permit the algorithms of the tools to be easily modified. We also do not consider the issue of data reorganization, as several database techniques are available for restructuring existing data after a change [Banerjee et al. 1987, Gerritsen & Morgan 1976, Lerner & Habermann 1990, Navathe & Fry 1976, Shu et al. 1977, Sockut & Goldberg 1979, Staudt 1988]. We do not discuss here the specific algorithms used to move data in and out of main memory; however, some of the techniques we present provide information that can make such movement more efficient. We assume that the SDE is implemented in potentially several languages, with these languages being compiled, strongly typed, and statically type-checked, for greater efficiency. Finally, we emphasize support for fine grained data, which we now justify.

1.2 Data Granularity

An SDE manipulates data across a wide range of granularity. At one end are very coarse objects, encoding information on large entities as a whole, such as projects, modules, development tasks, and requirement specifications [Penedo 1986, Tichy 1982, Wolf et al. 1989]. At the other end are very fine grained objects such as symbol table entries, statements in an abstract syntax tree, and procedure signatures [Clarke et al. 1986]. The performance requirements related to such data, relative to those concerning coarse grained objects, are quite severe. Fine grained objects are small, numerous, and tightly interconnected; they must be moved in and out of the tools with minimal overhead [Andrews & Harris 1987]. Consequently, fine grained data must be tightly bound, termed *impedance-matched*, to the language in which the tools are written to achieve adequate performance [Cockshott et al. 1984, Conradi et al. 1986, Straw et al. 1989]. That is, they must appear to the tool's code as instances of data types provided by the programming language. Objects are usually represented by records and object references by pointers, allowing navigation by pointer chasing, rather than via calls to a DBMS runtime library. Alternative language bindings, such as those available with conventional database systems, simply require too much overhead to navigate a graph or tree of objects, resulting in unacceptable performance, generally an order of magnitude slower than an impedance-matched representation.

While some work has addressed evolution of coarse grained data [Bernstein 1987, Dittrich et al. 1986, Garlan et al. 1986, Penedo et al. 1989, Skarra & Zdonik 1986, Skarra & Zdonik 1987], supporting evolution of fine grained data is an open research problem, and is the target of the strategies to be discussed shortly.

1.3 An Example

To illustrate the adverse effects of evolution, let us consider the ramifications of a specific change to an SDE. We decide to add a cross reference generator tool to this SDE. This tool will extract the symbol table of a selected module from the database, traverse it, and print out a formatted list of symbols along with the line number where each was declared. This change may be characterized as system evolution.

Unfortunately, the symbol table currently does not record source position information, as the other tools using the symbol table did not require that data. So we must also modify the symbol table structure (data specification evolution) and the lexical analysis tool, which should compute the source position (tool evolution). We decide to represent the source position as a single 16-bit integer storing the line number where the symbol occurred.

The impact of these changes is pervasive. First, all existing symbol tables stored in the database must be altered to add the new attribute, with a default value, say, a line number of 0, stored. Secondly, all tools that reference symbol tables must be modified. Due to an impedance-matched language binding, internal data structures used by each tool that encode symbol tables must be altered to include the new attribute. The tools must then be recompiled. Clearly, adding an attribute may be very costly.

Later we discover that our encoding is flawed: we have not allocated sufficient bits for the line number. So we increase the size of the source position attribute from 16 to 32 bits (representation evolution).

This impact of this seemingly minor change may be as pervasive as the original change. All existing source positions in the database must be updated: the new representation must be computed from the old and the data reformatted on disk, since an attribute has grown in size. All other tools that reference the symbol table must again be modified, even though they do not access the source position attribute. Hence, a minor change in representation of a little used attribute can also be costly, potentially generating several hours of data reorganization, code modification, and recompilation.

1.4 Previous Work

Several approaches have been proposed that can dramatically reduce the impact of seemingly minor changes, such as the example of representation evolution discussed above. One proposal is to apply the database concept of *views* [Date 1986, Ullman 1988] to SDE's. Each tool manipulates an identified subset, termed the view, of the database. The other side of the coin is the database may be characterized as the union of all of the tools' views [Garlan 1987]. Each tool is insulated from changes to portions of the database outside of that tool's view. For example, the semantic analyzer's view of the symbol table does not include a source position attribute, so the addition or change in representation of this attribute will not affect this tool.

Views effectively isolate the source code of the tool from many changes to the database; for this reason they are used frequently in conventional database applications. They are less successful when applied to fine grained data manipulated by an SDE. Views imply data conversion between the database and the tool. In the example given in the previous section, the source position attribute is present in a symbol table entry in the database, but is not present when that entry appears in the semantic analysis tool. If an attribute is added to an object, but that attribute is not part of the tool's view, then the source code for the tool need not change. However, the tool is impacted indirectly by this change. If the data conversion implied by the view is handled by table-driven DBMS runtime routines, then the tables need to be regenerated. Also, the runtime overhead imposed by table-driven conversion when reading and writing data is significant. If the conversion is handled by specialized code generated for each tool, then this code must be regenerated, compiled, and the tool relinked. In either case, additional overhead resulting from a change occurs when the data specifications for each tool that references the database are reanalyzed, a task that itself can take significant time. While a view may isolate the *logical* interface of a tool to changes to the database, such changes necessarily affect the code maintaining the *physical* interface.

The GRAPHITE system uses a related scheme, supporting two versions of the physical interface [Clarke et al. 1986]. The logical interface to fine grained objects consists of a set of routines that access and modify values of attributes. The *development version* provides a table-driven implementation of these routines, with similar advantages and drawbacks of views. The *production version* specifies inline expansion of these routines, so that attribute access expands into access via absolute offsets into the objects. The advantage is much higher execution-time performance; the drawback is significantly worsened development-time performance, as the entire tool must be recompiled on changes to the shared database.

The approach of providing development and production implementations underneath an identical logical interface is an excellent one. However, it falls short of providing a complete solution to achieving evolution resilience. First, it involves a binary decision, development or production version, while the environment developer would like finer control over the development-time/execution-time efficiency tradeoff. Second, the development version still requires significant work to return the SDE to an operational state. For the example above, when the source position attribute is changed, the database interface module for the semantic analyzer must be regenerated and compiled, and the semantic analyzer tool relinked. We desire an alternative that necessitates *no* changes to the semantic analysis tool when a source position attribute is added, even if execution-time performance suffers. Finally, this approach requires that a procedural interface to attribute access be used, which many find syntactically awkward and verbose.

Newcomer proposed that the previous representation be considered when computing the new representation after a change [Nestor et al. 1989, Newcomer 1986]. His approach applies to both tight language bindings and inline compiled accessor routines, such as those just discussed. The trick is to modify the representation in such a way that code accessing unaltered attributes is still correct. In the example, the source position attribute would be added to the *end* of the symbol table object. Then, all the code that accesses the symbol table, such as the semantic analysis tool, need not be altered, or even recompiled. Deleted attributes can be left in the object, and modified attributes can be treated as attributes that have been deleted and then later added.

The benefit of this approach is that SDE-development-time efficiency is enhanced. However, the resulting object layout may waste space, especially when classes and multiple inheritance are supported in the data model [Nestor et al. 1989]. Tools that create data objects must be relinked with a newly generated table that specifies the size of each object. More fundamentally, this approach is quite dangerous when applied manually, as tools can easily become inconsistent.

To summarize, previous work provides valuable techniques, yet does not support evolution resilience in a comprehensive fashion. For adequate SDE-execution-time performance, the representation of fine grained data *must* be tightly bound to each tool. Yet tight binding implies little or no resilience to change: the binding must be recomputed, often at significant cost at SDE-development-time, each time some aspect of the data or tools is modified. On the other hand, very loose binding, such as through table-interpreted views or accessor/modifier routines, which may achieve faster reorganization at SDE-development-time, will inevitably result in an inefficient SDE. Finally, even if the requirement for high performance were ignored completely, say, for the initial testing phases of the SDE, none of these approaches can accommodate a change without necessitating at least a relink of the effected tools, which may still imply a rather long minimal modify-recompile-test cycle.

1.5 Desiderata

We can summarize the problem by listing the requirements that must be met to achieve evolution resilience in an SDE. The rest of the paper will then integrate the previously proposed approaches discussed above with some new techniques to address these requirements.

- Each change should have as small an impact as possible. Ideally few if any tools not directly changed through tool or system evolution will need to be modified, recompiled, or even relinked. For instance, the semantic analyzer tool should be unaffected by either adding the source position attribute or changing its representation.

- No source level changes should be required of tools not specifically participating in tool or system evolution. If we must alter the executable of the semantic analysis tool when the source position attribute is added, such alteration must involve only recompilation or relinking. Involving the environment developer each time evolution occurs somewhere in the SDE is an invitation

for bugs and rapidly rising implementation costs as the SDE grows in number of tools and in complexity.

- Environment developer time should be minimized when restoring the SDE to operational status, which involves making the required changes to the source code, recompiling the tools, and restructuring existing data. For those tools that must be changed, relinking is preferable to recompiling, which is preferable to regenerating source code (for those components that are generated from other specifications, such as a grammar).

- Tool execution efficiency should not be compromised by approaches used to achieve evolution resilience. The semantic analyzer should not run slower after the source position attribute is added. In fact, the semantic analyzer should execute as fast as known optimizations permit.

- Consistency among the tools and the software database is paramount, and must be able to be guaranteed when requested by the environment developer. After the source position attribute is added, and various changes have been made, a support tool must be available to be used by the developer to at least check that consistency has again been achieved. Even better are support tools that ensure consistency at all times.

2 Approach

Unfortunately, there is no resolution to the tradeoff between SDE-development-time and SDE-execution-time efficiency when manipulating fine grained data [Wileden et al. 1990]. (We emphasize time efficiency here, but space efficiency is an important parallel issue.) No known techniques achieve both development-time and execution-time efficiency simultaneously, and none are anticipated. However, it is possible to exploit the different efficiency requirements present at various points during the development of an SDE. Specifically, for most of the time, rapid turnaround is paramount: when a change is made, a functioning SDE must be available rather quickly, so that productivity of the developer is maximized. SDE-execution-time efficiency is relatively less important, because test cases are generally small and because few tools are exercised by each test. As the evolving SDE matures, and as global changes, e.g., to the data base specification, become less prevalent, SDE-development-time efficiency is less important, because changes are more localized. However, at this point in the development cycle, SDE-execution-time efficiency becomes more of an issue, because test cases tend to be large and exercise many of the tools in the SDE. When the SDE is delivered, execution-time efficiency is virtually the only relevant metric; environment users will not be happy to hear that their SDE is slow because it was easier to build it that way!

Our approach has three components.

1. We define a *coupling spectrum*, consisting of a variety of approaches to tool interaction. Some of these strategies are very efficient, but are quite fragile, requiring a complete reanalysis on any change. Others are robust, yet are rather inefficient. Still others are intermediate in both SDE-development-time and SDE-execution-time efficiency. We introduce implementation techniques that support easy movement along the coupling spectrum anchored at one end by rapid SDE development and at the other end by efficient SDE execution. In particular, no source code need be changed when moving along alternatives within this spectrum.

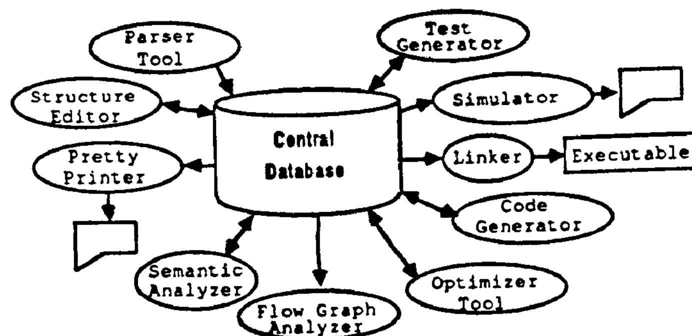

Figure 1: Central Database SDE Architecture

2. To provide precise control to the developer, we employ a *meta-environment*, a specific SDE tailored to the specification, design, implementation, and testing of *target* SDE's.

3. We use an *explicit tool topology* to describe the data shared among the tools, the tools themselves, and the static composition and interaction of the tools.

We next examine the need for an explicit tool topology and the support provided by the meta-environment. We then discuss the many points along the coupling spectrum.

3 Tool Interaction

The basic model of tool interaction employed in most SDE's, proposed or implemented, is the central database architecture shown in Figure 1. Tools (denoted with ovals) interact indirectly via information communicated through the database [Buxton 1980, Didriksen et al. 1987, Dittrich et al. 1986, Dowson 1987, Garlan 1987, Habermann & Notkin 1986, Hitchcock et al. 1986, Hudson & King 1988, Lewerentz 1988, Narayanaswamy & Scacchi 1987, Notkin 1985, Paseman 1989, Penedo 1986, Perry & Kaiser 1988, Reps & Teitelbaum 1989, Tichy 1982].

This model is not sufficiently expressive for the meta-environment to generate efficient code or to manage evolution of the developing SDE. We adopt a more refined model, in which tools communicate instances of strongly typed data structures. A tool topology is specified for the SDE, stating explicitly how the tools interact and what data is communicated. Figure 2 displays the topology of the SDE illustrated in Figure 1 (here, data structures are denoted with rectangles). Clemm and Osterweil have proposed a similar topology to automate tool invocation [Clemm & Osterweil 1990, Osterweil & Clemm 1983]; the two uses are consistent and orthogonal. The Polylith environment also includes this notion of tool integration [Kaplan, et al. 1986, Purtilo 1985, Purtilo 1988].

An explicit tool topology differs from the central database architecture in two ways. First, the central database is partitioned into multiple logical databases, with only a few tools interfacing to each of these smaller databases. The central database model implies more interaction than actually occurs; an explicit tool topology allows the ramifications of a specific change to be more precisely determined by the meta-environment. The second difference is that tool interaction need not be through the database. It is common, even in a tightly integrated SDE, for an output of a particular tool to be read by only one other tool. In such cases, the generality provided by a central database is not needed, and the performance degradation implied by that generality is not acceptable. Instead, these two tools can communicate through highly specialized interfaces that extract ultimate performance.

The presence of multiple databases brings up the issue of shared data. Some of the logical databases, such as one containing the attributed syntax tree of the standard prelude in an Ada environment, will be referenced by data in many of the other databases. The appropriate structuring and use of these databases is crucial to evolution resilience. One advantage of an explicit tool topology is that such databases are naturally emphasized in the topology, alerting the designer to their importance.

The (target) environment developer describes tool integration by formally specifying the data structures communicated among the tools. This specification is analogous to the schema written for and interpreted by a database management system. The developer then states, for each tool and database, where along the coupling spectrum that component should be placed. The meta-environment will automatically generate interface code and internal data structures in the implementation language of the tool, along with tables or code for the databases of the target SDE. The explicit tool connectivity allows the meta-environment to handle more of the details of tool connectivity, including computing representations for data structures, generating code tailored to the tool connectivity, organizing the structure of shared data, checking consistency between connections, and allowing the late binding of certain decisions. Automating these analytical and generative tasks is not possible without the meta-environment having access to the connectivity structure.

4 From Fast Evolution to High Performance

In this section we examine many points along the spectrum anchored at one end with high SDE-development-time efficiency (termed *fast evolution*) and at the other end with high SDE-execution-time efficiency (termed simply *high performance*). To characterize this spectrum, we utilize the concept of coupling of the representations of data structures internal to and passed between tools [Snodgrass & Shannon 1986]. At one end of the spectrum, *weak coupling*, a generic external representation is chosen and the tools must convert the internal data structure to and from this generic structure. With *strong coupling*, at the other end of the spectrum, the external representation closely mirrors that of the internal representation. Coupling, evolution, and performance are interrelated. In general, with stronger coupling the performance of the interacting tools is higher, but it is much more difficult to incorporate changes into a tool due to the heavy dependence between the representation of the internal and external data structures. Conversely, weak coupling promotes fast evolution but results in low performance due to the extra conversions required between data structures. In this section, we describe the two end points and many intermediate points along this spectrum and show how different selections affect evolution and performance. In each case, we describe the representation of the shared data written by one tool and read by another, thereby establishing a connection between the two tools. Each connection present in the tool topology of the (target) SDE may be independently positioned along the coupling spectrum, thereby affording the developer precise control over the fast evolution/high performance tradeoff.

For each connection:
- Content
- Representation
- Consistency: Repacking,
  - Consistency Maintenance
- Form: ASCII External Representation,
  - Independent Binary,
  - Dependent Binary,
  - Incremental I/O,
  - Shared Memory

For each tool:
- Content
- Representation
- Consistency: Repacking,
  - Consistency Maintenance
- Physical Interface: Impedance-matched,
  - Procedural
- Binding: Compile-time,
  - Link-time,
  - Runtime

Figure 3: Developer Decisions

For each connection between two interacting tools, the developer must make four decisions, as summarized in Figure 3. The first is the *content* of the shared data, that is, the objects and attributes to be included. The second decison is the *representation* of this data, e.g., the number of bits required to encode an integer value. The third decision is the *consistency* of the representation of the shared data. The default is *repacking*, where the previous representation is not considered in determining various low level details such as attribute order within an object. The alternative is *consistency maintenance*, where the low level details are chosen so that previous data will still be acceptable to the tool, if possible. A related, but distinct, technique is *lazy reorganization*, where a portion of a database is reorganized upon first access [Banerjee et al. 1987, Lerner & Habermann 1990]. Lazy reorganization offers a partial solution when reorganization of the data is not possible, as with backed-up data, data archived on read-only optical disk, or data for which write permission is not available. The final decision is the *form* of the data, of which there are five alternatives located along the coupling spectrum.

At the weakest end of the spectrum, tools interact via an *ASCII external representation* of the shared data. The ASCII representation is machine independent as well as independent of the language of the interacting tools. With this form of data, changing the representation of a type within

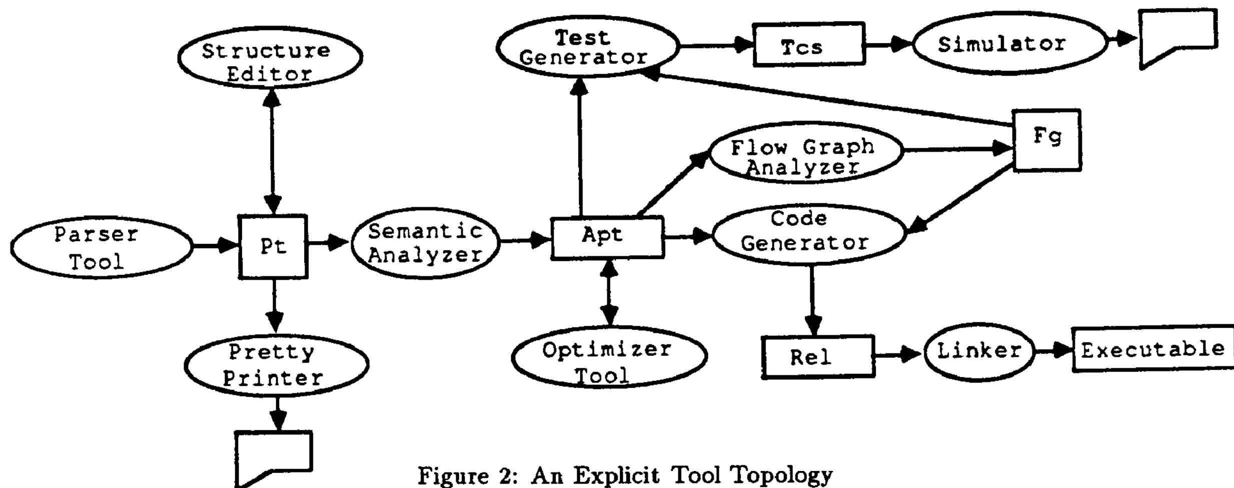

Figure 2: An Explicit Tool Topology

one tool does not affect the external representation of the data or the representation of the data within the second tool, thus allowing representation changes to remain localized within a tool. Adding an attribute or object within one tool will not affect either the external representation nor the second tool. If a tool does not use an attribute or object type of a data structure that is read in, that attribute or object type may be removed without needing to modify or even recompile the tool. If an attribute is removed from the database, and the tools that previously used it are modified to no longer use it, then no existing data nor other tools need be modified; that attribute will simply be ignored during input.

Since changes to the representation require only a possible recompilation of the affected tool, this localization allows for fast evolution of tools. The price is poor SDE-execution-time performance due to the extensive conversions required on input and output. To write the data, the producing tool converts each internal data structure into a formatted ASCII description. To read the data, the receiving tool must parse the ASCII representation of the data and map each object into its corresponding data structure in main memory. Finally, external data instances in this representation are rather large, as each object and attribute is named with a text string each time it is mentioned, and values of basic types such as integers are also given as text strings. The performance of these conversions is therefore poor in terms of both time and space.

An improvement in performance is gained by replacing the ASCII external representation of the shared data with an *independent binary representation*. This representation is also machine and language independent but is more strongly coupled with the internal representation of the data [Lamb 1987A]. Here, object and attribute names can be encoded in one byte, integers and rationals generally in 4 or 8 bytes, and boolean values potentially in one bit. Some changes to the specification or representation of internal data may affect the conversion code within the both tools. For instance, if one attribute is removed, the names of the remaining attributes might be assigned different byte encodings. Other type representation changes, such as adding an attribute within a single tool, will remain localized. The amount of work for tool evolution is therefore slightly higher than with the ASCII external representation. The benefit is increased performance in the conversions between external and internal representations of the data. Although it is still necessary to convert between the independent binary representation of the external data and the language dependent internal representation of the internal data, the parsing or formatting of the external data is much simpler because the two representations of the data are more similar than with the ASCII representation.

Further improvements in performance are possible by using a *dependent binary representation* for external data which is neither machine or language independent. The external representation of the data is effectively a slightly altered core dump of the internal data representation, with two alterations: pointers are virtualized and some attributes may be omitted [Newcomer 1987]. Input and output are thus more efficient than previously described techniques, especially when the internal and external representations contain the same attributes. The external and internal representations are therefore very strongly coupled with the result that evolution requires more work. Changes to objects or their representations in one tool are guaranteed to affect the representation in the second tool.

In the three previous described representations, we assumed that all possible accessible objects are read at tool initialization. The advantages are that no cache residency checks (that determine whether a referenced object is actually in main memory) are required, and disk I/O requires few seeks since data is read sequentially. The disadvantages are excessive main memory usage and excessive I/O. A performance improvement is obtained by using *incremental I/O*, where only a subset of the reachable objects are cached in main memory. In one approach, termed *object faulting*, a single object is read into main memory when it is first referenced; all subsequent references to this object are converted into pointers [Cockshott et al. 1984, Wileden et al. 1988]. A more efficient approach is *clustering*, where the unit of data transfer is a *segment* containing multiple objects [Andrews & Harris 1987, Hornick & Zdonik 1987, Krueger et al. 1989, Shannon & Snodgrass 1990, Stamos 1984]. The advantage is that I/O is more efficient since segments are generally a multiple of the page size and only relevant segments are retrieved. This advantage depends on an assumed correlation of *temporal locality* and *spatial locality*; objects referenced together in time reside together in the same segment. The main disadvantages are that each object fault interrupts the tool and causes a disk seek and some cache residency checks are required. However, proper clustering can minimize cache residency checks and object faults and subsequently reduce disk seeks. With clustering, small changes to one tool's data

representation or clustering definitely affect the external representation and the second tool's representation.

At the farthest end of the coupling spectrum, two tools interact via *shared memory*. The tools may be in separate processes, if the operating system supports memory sharing by disparate processes, or the tools may reside in a single process. In this form of data, the tools do not actually read or write data instances; the data instances reside only in main memory and the tools simply pass pointers. The advantage is greater performance since I/O between the tools is eliminated. The disadvantage is the dependence between the tools' internal representations results in extensive changes if one tool modifies a type representation.

As also shown in Figure 3, the developer must make five decisions for each tool in the SDE. The first three are the *content* of the internal data manipulated in main memory by the tool, the *representation* of this data, and the *consistency* of the main memory representation. The fourth is the *physical interface* to the object residing in main memory (we advocate a particular *logical interface*, visible to the programmer, in Section 6.2). There are two alternatives, an *impedance-matched* interface, in which attributes are accessed directly using the target language constructs, and a *procedural* interface, in which attributes are manipulated through routines generated for that purpose. The last decision to be made for each tool is the *binding* of the code that performs data I/O and manipulates attribute values. The alternatives are *compile-time* binding, which implies hard code for I/O and inline attribute manipulation routines, *link-time* binding, which implies table-driven I/O, with the tables implemented as static data initialized in a separately-compiled module that is linked with the rest of the tool's code, and attribute manipulation routines not inlined, but placed in a separately-compiled module, and *runtime* binding, which is similar except that the code is dynamically linked into the running executable when the first routine call is made. With the table-driven approach (link-time and runtime binding), there is no need to regenerate the reader and writer of the external data structure on a change; only the table describing the representation need be regenerated and the program relinked. Even less work is required for runtime binding to return the tool to an operational state after a change. With compilation-time binding, all code for tool I/O is regenerated. As one might expect, the reduction in evolution flexibility is offset by an increase in performance over the table-driven approach.

Alternatives for each of the decisions just discussed may be combined in various ways, yielding the coupling spectrum shown in Figure 4. The ordering among options is approximate. Some options are actually collections of related options. For instance, there are many approaches to clustering (e.g., clustering by object type, dynamic adaptation, syntactic clustering, semantic clustering, fragmentation [Shannon & Snodgrass 1990]), all grouped under options 21 and 22. Hence, there are more points along the spectrum than shown in Figure 4, though any meta-environment would of course support only a subset.

Some combinations of physical interface and binding do not make sense. For example, a procedural physical interface with a compile-time binding is not shown, because this combination would be slightly less efficient than an impedance-matched interface with compile-time binding, while not exhibiting any greater evolution resilience, and so would be dominated by the latter combination.

In general, the approaches with weaker coupling between tools (at the top of the figure) are more advantageous for fast evolution since changes are more localized within a tool. As the tools evolve toward a more stable state, approaches with stronger coupling (towards the bottom of the figure) are preferable due to higher performance. Note that the last six options in the coupling spectrum have a compile-time binding. In cases where recompilation of existing tools is infeasible (say, once the tool has been delivered to users), the developer will be restricted to linktime and runtime bindings, and option 14 or 17 may yield the highest feasible performance.

5 An Example

To illustrate the tradeoff between fast evolution and high performance, we present the implementation of a programming environment and examine several points in the lifecycle of the system. While this environment is simplistic, it nevertheless shares many aspects with larger, more realistic SDE's. The environment initially contains three tools, a parser, a semantic analyzer, and an optimizer. The parser reads in a textual description of a program and produces a syntax tree. The semantic analyzer performs name resolution and type analysis on the syntax tree and outputs a directed attributed graph. Finally, the optimizer reads the attributed graph, performs constant folding, and outputs a potentially more efficient version of the attributed graph. The initial configuration is shown in Figure 5. We summarize the changes made over a period of months as this simple SDE is implemented. At various points, some aspect becomes more or less stable, and a more efficient interface is adopted, so that the SDE executes faster and individual test cases take less time to run.

During the early development, the specification of the data evolves resulting in frequent changes to the shared data (in A, B, and C) in the form of object and attribute additions and deletions. Also, the algorithms for all three tools are in flux. To ensure fast evolution, we specify that the parser and the semantic analyzer communicate via the ASCII external representation. One advantage of this representation is that data instances are easily created and viewed with a text editor, simplifying debugging. The internal data structure of the parser will consist of the syntax tree plus additional attributes and object types. Since changes to the syntax tree will probably require source level changes to the parser anyway, we specify an impedance-matched interface, with static table-driven output for quick regeneration of the writer (option 4). The same decisions are made for the semantic analyzer and the optimizer; additionally, they employ table-driven input. The ASCII external representation allows fast evolution of the tools since changes to the tools are more localized requiring fewer recompilations. For instance, adding an internal attribute in the semantic analyzer to aid processing will not affect the parser or optimizer at all. Performance is slow but is low priority at this point in the lifecycle.

Further along in the development, the specification of the syntax tree stabilizes but the representation of the data as well as the specification of the attributed graph and of the internal data structures continue to evolve. We modify the form of A to an independent binary representation to allow bigger examples to be run, and specify that changes are to be consistent with the previous representation. At this time the specification and representation of the data communicated between the semantic analyzer and the optimizer continue to evolve, thus the external representation between those two

#	Form	Physical Interface	Binding	Consistency
1	ASCII External Representation	Procedural	Runtime	—
2	ASCII External Representation	Procedural	Link-time	—
3	ASCII External Representation	Impedance-matched	Runtime	—
4	ASCII External Representation	Impedance-matched	Link-time	—
5	ASCII External Representation	Impedance-matched	Compile-time	—
6	Independent Binary	Procedural	Runtime	Consistent
7	Independent Binary	Procedural	Link-time	Consistent
8	Independent Binary	Procedural	Runtime	Repacking
9	Independent Binary	Procedural	Link-time	Repacking
10	Independent Binary	Impedance-matched	Runtime	Consistent
11	Independent Binary	Impedance-matched	Link-time	Consistent
12	Independent Binary	Impedance-matched	Compile-time	Consistent
13	Independent Binary	Impedance-matched	Runtime	Repacking
14	Independent Binary	Impedance-matched	Link-time	Repacking
15	Independent Binary	Impedance-matched	Compile-time	Repacking
16	Dependent Binary	Impedance-matched	Link-time	Consistent
17	Dependent Binary	Impedance-matched	Link-time	Repacking
18	Dependent Binary	Impedance-matched	Compile-time	Repacking
19	Incremental: Object Faulting	Impedance-matched	Compile-time	Consistent
20	Incremental: Object Faulting	Impedance-matched	Compile-time	Repacking
21	Incremental: Clustering	Impedance-matched	Compile-time	Consistent
22	Incremental: Clustering	Impedance-matched	Compile-time	Repacking
23	Shared Memory	Impedance-matched	Compile-time	Consistent
24	Shared Memory	Impedance-matched	Compile-time	Repacking

Figure 4: The Coupling Spectrum

tools remains as ASCII. The table-driven I/O of all three tools remain, implying option 11 for the parser and for the input size of the semantic analyzer, and option 4 for the output side of the semantic analyzer and for the optimizer. With this arrangement, a change to the attributed graph structure will not affect the parser at all. A change to the syntax tree will be quite costly in terms of SDE-development time; we're bargaining that such changes are rare, so that our test cases run faster.

Later, we decide to add a new tool, a cross referencer, to the environment. The result of this system evolution is depicted in Figure 6. We need to change both the syntax tree and attributed graph structures to add a source position attribute, and to change the parser tool to compute this new attribute. Since we specified consistent changes, this attribute will be added at the very end of the symbol table object. The semantic analysis and optimizer tools will need to be relinked, but not recompiled, thus reducing SDE-development time to effect this change. The cross referencer will utilize an impedance-matched binding, with static table-driven input, on the assumption that the specification of the shared syntax tree structure is relatively stable. Since we are still primarily testing the parser-semantic analysis-optimizer pipeline, we *don't* specify consistent changes with respect to the cross referencer (option 14).

Still later, when we increase the size of the source position attribute, discussed earlier in Section 1.3, only the parser and the semantic analyzer need be changed.

The optimizer is debugged more quickly than the semantic analyzer, and stabilizes about the same time as the attributed graph structure. The semantic analyzer, probably the most complex tool, continues to evolve. The optimizer continues to read the ASCII external representation output by the semantic analyzer but now outputs the optimized attributed graph in dependent binary representation using hard-wired attributes. Evolution now consists primarily of changes to the semantic analyzer algorithm and modifications to internal attributes within this tool. The parser and optimizer are unaffected by these changes. Since the parser is now very stable, we switch its output to the dependent binary representation, with hardwired output code (option 18). We do the same for the semantic analyzer. We specify that attributes are to be packed across a change, rather than attempting to avoid recompilation. Changes will be rare, and we want the I/O to go relatively fast.

Finally, the semantic analyzer is stable and performance becomes a high priority. The first three tools are now merged into a single process, as shown in Figure 7. The syntax tree and attributed graph now reside in main memory with the tools simply passing pointers. Internal attributes for the three tools are placed in the main memory objects (option 24). The optimizer outputs the optimized attributed graph with clustered I/O to a single database, which is read in a clustered fashion by the cross referencer. To make the cross referencer as fast as possible, the attributed graph is divided into two clusters: the portion needed by the cross referencer and the remaining portion. The writer for the optimizer is configured to output this clustered version of the attributed graph, and the reader for the cross referencer is modified to only access the relevant segments (option 22). The performance of the system is now at its highest, and the SDE-development-time efficiency is at its lowest. Any change to any aspect, whether changing one of the external data structures such as the syntax tree, changing an attribute used internally by a tool, or changing the clustering, will generally require regenerating the input and output routines and internal data structures and recompiling all four tools in the SDE.

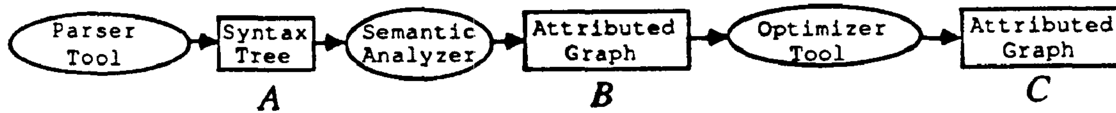

Figure 5: Initial Configuration

6 Meta-Environment Support

The last section presented a scenario where individual tools move along the coupling spectrum during the development of an SDE. We now discuss how the meta-environment assists the environment developer in specifying this example. Much of this meta-environment has been implemented, and is in daily use in the development of several SDE's by various academic and commercial research organizations [Snodgrass 1990]. The meta-environment contains approximately 15 tools, one containing seven basic tools, for a total of approximately 100K lines of source code. Our implementation currently supports options 4, 17, and 24 of Figure 4; we are now implementing the independent binary representation, incremental I/O (using a promising technique termed *semantic clustering* [Shannon & Snodgrass 1990]), and consistency maintenance, which will provide an additional 10 options.

6.1 Specifying Tool Topology and Evolution

Initially, the tool developer describes the parser, semantic analyzer, and optimizer tools using a high level specification language [Snodgrass 1989]. The tool developer first describes the data shared among tools utilizing *structures*, or collections of object and attribute declarations, as a linguistic device to aid in the description. Structures can be derived from other structures in various ways permitting sets of similar data to be described in terms of each other. Structures can also be refined from other structures allowing further constraints such as representational aspects to be placed on the data. For the programming environment example, the tool developer writes a structure for the syntax tree and then derives from this an additional structure for the attributed graph. The attributed graph is conveniently described in terms of the syntax tree since differences consist of additional attributes.

The next step for the tool developer is to describe the three tools using the specification language. Basically, a tool specification describes the flow of data in a tool. Therefore, tools are described in terms of the data they produce and consume. *Ports* specify the data structures read and written by the tool. Ports are typed; associated with each port is a structure. Input ports read an instance of the structure into main memory and return the root of the instance to the algorithm. Output ports write from main memory the instance referenced by the root object passed as a parameter. In the programming environment example, the parser has one output port which produces an syntax tree. The semantic analyzer has one input and one output port; the input port reads a structure of type syntax tree and the output port writes a structure of type attributed graph. Similarly, the optimizer tool also has one input and one output port, both of type attributed graph. The developer also defines the language in which the tool is implemented, as well as the structure and low-level representation of the data in main memory that the tool's algorithm uses.

The third task for the tool developer is to specify the interaction of the tools. Tool interaction is specified using a linguistic construct called *connections*. This construct is used to connect input and output ports of communicating tools. In our example, one connection is used between the parser and semantic analyzer and a second connection is used between the semantic analyzer and the optimizer.

We extended the Interface Description Language (IDL) [Nestor et al. 1982], which already included structures, tools, derivation, refinement, and ports (and which was definitely *not* evolution resilient), with linguistic constructs to specify connections, representations on connections, language bindings, databases, and clustering. The added constructs allow the developer to specify the complete tool topology, to choose among intermediate points in performance, and to be explicit about what data is to be made persistent.

Starting with IDL had two benefits. IDL is language-independent, allowing tools to be implemented in different languages. Also, it was designed to express those data structures used in a programming environment, such as parse trees and symbol tables. In particular, it supports a sophisticated type model employing a class hierarchy, multiple superlasses, inheritance, and set and sequence collection types [Shannon & Snodgrass 1989]. It also has an associated assertion language. However, nothing in our approach precludes application in language-specific contexts, such as an environment implemented entirely in C++, provided adequate meta-environment support, discussed in the next section, and expressive means to specify tool topology are available.

The analysis of structure, tool, connection, representation, and clustering specifications is performed by the language specification translator. The translator is also responsible for generating code for attribute access, attribute modification, object creation, object faulting, and object storage and retrieval in the database. In this way, we achieve *specification level interoperability* by providing all four required components: a unified type model, language bindings, an underlying implementation, and automated assistance [Wileden et al. 1989].

6.2 Isolating the Source Code

As the tool developer utilizes the various linguistic constructs to tune the performance of the system, it is imperative that different representation and clustering alternatives do not affect the source code of the tools being developed [Clarke et al. 1986]. One approach is to use inline routines or macros for access and modification of attributes. These routines can be generated automatically by the translator. The advantages of using routines are that they hide the representational details of attributes and they support clustering, since memory residency can be checked in the expanded code. The disadvantage is that the routines are syntactically awkward to use, especially when accessing through several levels of indirection. A second disadvantage is the overhead

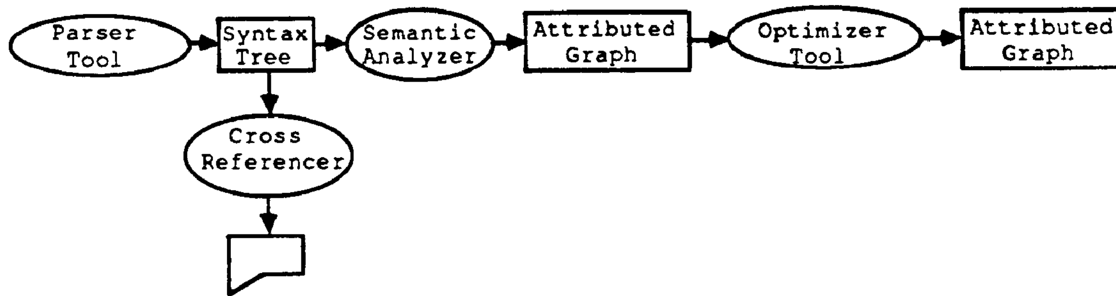

Figure 6: A Tool is Added

imposed on the target language compiler due to the *big inhale* of the large number of generated routines during compilation [Conradi & Wanvik 1985].

Our approach is to instead use a *preprocessor* that inserts the appropriate code for all aspects of data access and modification, including any memory residency checks. An advantage is that the syntax is cleaner; the construct provided by the language for attribute access in a record can be used regardless of the implementation. In addition, there is less overhead for the language compiler since it does not need to contend with the many generated routines. Finally, the opportunity exists to have very sophisticated representations for attributes. For example, some attributes can be stored outside of the object in main memory [Lamb 1987B]. The preprocessor generates the necessary code to access the external attribute.

The preprocessor can support both physical interface alternatives. For the C language construct used in the source code of the tool,

```
AnObject.attrname
```

the preprocessor might emit [Shannon & Snodgrass 1989]

```
AnObject.IDLclassCommon->attrname
```

for the impedance-matched interface, and

```
getattribute(AnObject,"Object","attrname")
```

for the procedural interface. In this second case, the body of `getattribute` would interpret an internal table describing the layout of the object when it resides in main memory. In all cases, the *logical* interface seen by the programmer remains familiar language constructs.

It should be noted that other tools in the meta-environment, such as debuggers and cross referencers, must be cognizant of both the logical and physical interface. As an example, we have implemented a tool that works with the debugger to display data instances [Cook 1988]. This tool can display at an abstract, language-independent level, as well as the bit encoding of the data.

The meta-environment provides the tool developer with the ability to move easily among alternatives during evolution of the system. Linguistic constructs are provided to specify the tool topology, structure representation, and clustering. The analysis of the constructs is performed by a language specific translator in the meta-environment. The use of a special preprocessor provides the control necessary for all aspects of data access, modification, and I/O. These meta-environment features conspire to achieve evolution resilience of the developing SDE.

7 Conclusions

Wileden, Clarke, and Wolf have identified three basic techniques for defining shared data within an SDE [Wileden et al. 1990]. After an extensive evaluation, they concluded that one technique, specification-described, was superior to the other two techniques, implementation-described and value-described, in terms of consistency management and development and reuse effort, and inferior in terms of SDE-development-time efficiency (their term was turnaround time). Our approach, as described in Section 6.1, may be characterized as specification-described. In this paper, we showed how it is possible to initially sacrifice SDE-execution-time efficiency to achieve SDE-development time efficiency. We also explained how to shift the SDE, with no changes to its source code, to many intermediate points along the coupling spectrum, eventually reaching the other end late in the development, thereby achieving high SDE-execution-time efficiency. We introduced new techniques at both ends of the spectrum: the ASCII external representation with a procedural interface for achieving fast evolution, and semantic clustering and shared memory for achieving high tool execution performance. Finally, we discussed our implementation of a meta-environment that interprets an explicit tool topology describing the data shared among tools in the target SDE and the static composition and interaction of the tools, thereby providing precise control to the developer.

We conclude that evolution resilience *is* achievable in an SDE without jettisoning other desirable properties such as high performance, ease of understanding, ease of change, reuse, consistency management, or controlling the impact of change.

8 Acknowledgement

The referees made several helpful comments that improved this paper.

9 Bibliography

[Andrews & Harris 1987] Andrews, T. and C. Harris. *Combining Language and Database Advances in an Object-Oriented Development Environment*, in *OOPSLA Conference Proceedings*. Association for Computing Machinery. Dec. 1987, pp. 430-440.

[Banerjee et al. 1987] Banerjee, J., W. Kim, H.-J. Kim and H.F. Korth. *Semantics and Implementation of Schema Evo-*

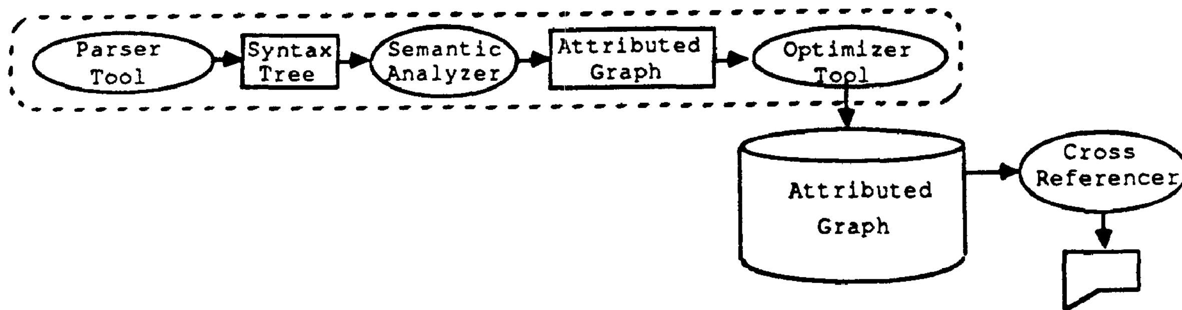

Figure 7: A Faster Version of the Example SDE

lution in Object-Oriented Databases, in *Proceedings of ACM SIGMOD International Conference on Management of Data*. Ed. U. Dayal and I. Traiger. Association for Computing Machinery. San Francisco, CA: 1987, pp. 311-322.

[Barstow et al. 1984] Barstow, D.R., H.E Shrobe and E. Sandewall. *Interactive Programming Environments*. McGraw-Hill Book Company, 1984.

[Bernstein 1987] Bernstein, P. A. *Database System Support for Software Engineering- An Extended Abstract*, in *Ninth International Conference on Software Engineering*. IEEE, ACM. Monterey, CA: Computer Society Press, Mar. 1987, pp. 166-178.

[Buxton 1980] Buxton, J.N. *Requirements for Ada Programming Support Environments-'Stoneman'*. Technical Report. Department of Defense. Feb. 1980.

[Clarke et al. 1986] Clarke, L.A., J.C. Wileden and A.L. Wolf. *Graphite: A Meta-Tool for Ada Environment Development*, in *Proceedings of the International Conference on Ada Applications and Environments*. Miami Beach, FL: IEEE Computer Science Press, Apr. 1986, pp. 81-90.

[Clemm & Osterweil 1990] Clemm, G. and L. Osterweil. *A Mechanism for Environment Integration. ACM Transactions on Programming Languages and Systems*, 12, No. 1, Jan. 1990, pp. 1-26.

[Cockshott et al. 1984] Cockshott, W., M. Atkinson, K. Chisholm, P. Bailey and R. Morrison. *Persistent Object Management Systems. Software-Practice and Experience*, 14 (1984), pp. 49-71.

[Conradi & Wanvik 1985] Conradi, R. and D.H. Wanvik. *Mechanisms and Tools for Separate Compilation*. Technical Report 25/85. The University of Trondheim, The Norwegian Institute of Technology. Oct. 1985.

[Conradi et al. 1986] Conradi, R., T. Didriksen and A. Lie. *IDL as a Data Description Language for a Programming Environment Database*. EPOS 15. Division of Computer Science, University of Trondheim. July 1986.

[Cook 1988] Cook, R.E. *A Tool for Viewing IDL Data Structures*. M.S. Thesis. Computer Science Department, University of North Carolina at Chapel Hill. Apr. 1988.

[Date 1986] Date, C.J. *An Introduction to Database Systems*. Vol. I of Addison-Wesley Systems Programming Series. Reading, MA: Addison-Wesley Pub. Co., Inc., 1986.

[Didriksen et al. 1987] Didriksen, T., A. Lie and R. Conradi. *IDL as a Data Description Language for a Programming Environment Database. SIGPlan Notices*, 22, No. 11, Nov. 1987, pp. 71-78.

[Dittrich et al. 1986] Dittrich, K.R., W. Gotthard and P.C. Lockemann. *DAMOKLES- A Database System for Software Engineering Environments*, in *Proceedings of the International Workshop on Advanced Programming Environments*. IFIP WG2.4. Trondheim, Norway: June 1986, pp. 345-259.

[Dowson 1987] Dowson, M. *Integrated Project Support with IStar. Software*, 4, No. 6, Nov. 1987, pp. 6-15.

[Garlan et al. 1986] Garlan, D., C.W. Krueger and B.J. Staudt. *A Structural Approach to the Maintenance of Structure-Oriented Environments*, in *SIGSoft/SIGPlan Software Engineering Symposium on Practical Software Development Environments*. Association for Computing Machinery. Palo Alto, CA: SIGPlan, Dec. 1986, pp. 160-170.

[Garlan 1987] Garlan, D. *Views for Tools in Integrated Environments*. PhD. Diss. Computer Science Department, Carnegie-Mellon University, May 1987.

[Gerritsen & Morgan 1976] Gerritsen, R. and H.L. Morgan. *Dynamic Restructuring of Databases with Generation Data Structures*, in *Proceedings of the ACM Annual Conference*. Association for Computing Machinery. Houston, TX: Oct. 1976, pp. 281-286.

[Habermann & Notkin 1986] Habermann, A.N. and D. Notkin. *Gandalf: Software Developement Environments. Transactions on Software Engineering*, SE-12, No. 12, Dec. 1986, pp. 1117-1127.

[Hitchcock et al. 1986] Hitchcock, P., A.W. Brown, R. Weedon, A.N. Earl, R.P. Whittington and D.S. Robinson. *The Use of Databases for Software Engineering*, in *Proceedings of the Fifth British National Conference on Databases-BNCOD 5*. Ed. E.A. Oxborrow. University of Kent. Canterbury, England: July 1986.

[Hornick & Zdonik 1987] Hornick, M.F. and S.B. Zdonik. *A Shared, Segmented Memory System for an Object-Oriented Database. ACM Transactions on Office Information Systems*, 5, No. 1 (1987), pp. 70-85.

[Hudson & King 1988] Hudson, S. and R. King. *The Cactis Project: Database Support for Software Engineerings. IEEE Transactions on Software Engineering*, 14, No. 6, June 1988.

[Kaplan, et al. 1986] Kaplan, S.M., R.H. Campbell, M.T. Harandi, R.E. Johnson, S.N. Kamin, J.W.S. Liu and J.M. Purtilo. *An Architecture for Tool Integration*, in *Proceedings of the International Workshop on Advanced Programming Environments.* IFIP WG 2.4. Trondheim, Norway: June 1986, pp. 109-124.

[Krueger et al. 1989] Krueger, C.W., B.J. Staudt and A.N. Habermann. *Scaling Up Integrated Software Development Environment Databases*, in *Proceedings of the 1989 ACM SIGMOD Workshop on Software CAD Databases.* Ed. L.A. Rowe and S. Wensel. Napa, CA: Feb. 1989, pp. 74-78.

[Lamb 1987A] Lamb, D.A. *IDL: Sharing Intermediate Representations. ACM Transactions on Programming Languages and Systems*, 9, No. 3, July 1987, pp. 297-318.

[Lamb 1987B] Lamb, D.A. *Implementation Strategies for DIANA Attributes. SIGPlan Notices*, 22, No. 11, Nov. 1987, pp. 44-54.

[Lerner & Habermann 1990] Lerner, B.S. and A.N. Habermann. *Beyond Schema Evolution to Database Reorganization*, in *Proceedings of ECOOP/OOPSLA '90.* Ottawa, Canada: 1990.

[Lewerentz 1988] Lewerentz, C. *Extended Programming in the Large in a Software Development Environment*, in *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* Ed. Peter Henderson. Boston, MA: Nov. 1988, pp. 173-182.

[Narayanaswamy & Scacchi 1987] Narayanaswamy, K. and W. Scacchi. *A Database Foundation to Support Software System Evolution. Journal of Systems & Software*, 7, No. 1 (1987), pp. 37-49.

[Navathe & Fry 1976] Navathe, S.B. and J.P. Fry. *Restructuring for Large Databases: Three Levels of Abstraction. ACM Transactions on Database Systems*, 1, No. 2, June 1976, pp. 138-158.

[Nestor et al. 1982] Nestor, J.R., W.A. Wulf and D.A. Lamb. *IDL - Interface Description Language - Formal Description - Draft Revision 2.0.* Internal Document. Computer Science Department, Carnegie Mellon University. June 1982.

[Nestor et al. 1989] Nestor, J.R., J.M. Newcomer, P. Giannini and D. Stone. *IDL: The Language and Its Implementation.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

[Newcomer 1986] Newcomer, J.M. *IDL: Past Experience and New Ideas*, in *Proceedings of the International Workshop on Advanced Programming Environments.* IFIP WG 2.4. Trondheim, Norway: June 1986.

[Newcomer 1987] Newcomer, J.M. *Efficient Binary I/O of IDL Objects. SIGPlan Notices*, 22, No. 11, Nov. 1987, pp. 35-43.

[Notkin 1985] Notkin, D. *The GANDALF Project. Journal of Systems and Software*, 5, No. 2, May 1985, pp. 91-106.

[Osterweil & Clemm 1983] Osterweil, L. and G. Clemm. *The Toolpack/IST Approach To Extensibility In Software Environments*, in *Lecture Notes in Computer Science Ada Software Tools Interfaces.* Ed. G. Goos and J. Hartmanis. Workshop, Bath : Springer-Verlag, 1983, pp. 133-163.

[Paseman 1989] Paseman, W. *Architecture of the Atherton Software BackPlane*, in *Proceedings of the 1989 ACM SIGMOD Workshop on Software CAD Databases.* Ed. L.A. Rowe and S. Wensel. Napa, CA: Feb. 1989, pp. 105-108.

[Penedo 1986] Penedo, M.H. *Prototyping a Project Master Database for Software Engineering Environments*, in *Second Software Engineering Symposium on Practical Software Development Environments.* ACM SigSoft/SigPlan; ONR. Palo Alto, CA: Dec. 1986.

[Penedo et al. 1989] Penedo, M.H., E. Ploedereder and I. Thomas. *Object Management Issues for Software Engineering Environments. SIGPlan Notices*, 24, No. 2, Feb. 1989, pp. 226-231.

[Perry & Kaiser 1988] Perry, D.E. and G.E. Kaiser. *Models of Software Development Environments*, in *Proceedings of the International Conference on Software Engineering.* Raffles City, Singapore: Apr. 1988.

[Purtilo 1985] Purtilo, J.M. *Polylith: An Environment to Support Management of Tool Interfaces*, in *Proceedings of the ACM SIGPlan '85 Symposium on Language Issues in Programming Environments.* Seattle, WA: July 1985, pp. 12-18.

[Purtilo 1988] Purtilo, J.M. *A Software Interconnection Technology.* Technical Report UMIACS-TR-88-83, CS-TR-2139. Institute for Advanced Computer Studies, Department of Computer Science. Nov. 1988.

[Reps & Teitelbaum 1989] Reps, Thomas W. and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language Based Editors.* Springer-Verlag, 1989.

[Schwanke & Kaiser 1988] Schwanke, R.W. and G.E. Kaiser. *Smarter Recompilation. ACM Transactions on Programming Languages and Systems*, 10, No. 4, Oct. 1988, pp. 627-632.

[Shannon & Snodgrass 1989] Shannon, K.P. and R. Snodgrass. *Mapping the Interface Description Language Type Model into C. IEEE Transactions on Software Engineering,* 15, No. 11 (1989), pp. 1333-1346.

[Shannon & Snodgrass 1990] Shannon, K.P. and R. Snodgrass. *Semantic Clustering,* in *Proceedings of the Workshop on Persistent Object Systems.* Martha's Vineyard, MA: Sep. 1990.

[Shu et al. 1977] Shu, N.C., B.C. Housel, R.W. Taylor, S.P. Ghosh and V.Y. Lum. *EXPRESS: A Data EXtraction, Processing, and REStructuring System. ACM Transactions on Database Systems,* 2, No. 2, June 1977, pp. 134-174.

[Skarra & Zdonik 1986] Skarra, A.H. and S.B. Zdonik. *The Management of Changing Types in an Object-Oriented Database,* in *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications.* Ed. N. Meyrowitz. Association for Computing Machinery. Portland, OR: Nov. 1986, pp. 483-495.

[Skarra & Zdonik 1987] Skarra, A.H. and S.B. Zdonik. *Type Evolution in an Object-Oriented Database,* in Research Directions in Object-Oriented Programming. of Computer Systems Series. Cambridge, MA: MIT Press, 1987. Chap. Part 3. pp. 393-415.

[Snodgrass & Shannon 1986] Snodgrass, R. and K.P. Shannon. *Supporting Flexible and Efficient Tool Integration,* in *Proceedings of the International Workshop on Advanced Programming Environments.* IFIP WG 2.4. Trondheim, Norway: Springer-Verlag, June 1986, pp. 290-313.

[Snodgrass 1989] Snodgrass, R. *The Interface Description Language: Definition and Use.* Rockville, MD: Computer Science Press, 1989.

[Snodgrass 1990] Snodgrass, R. *IDL Toolkit Release 4.2.* Department of Computer Science, University of Arizona, Tucson, AZ, 1990.

[Sockut & Goldberg 1979] Sockut, G.H. and R.P. Goldberg. *Database Reorganization - Principles and Practice. ACM Computing Surveys,* 11, No. 4, Dec. 1979, pp. 371-395.

[Stamos 1984] Stamos, J.W. *Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory. ACM Transactions on Computer Systems,* 2, No. 2, May 1984, pp. 155-180.

[Staudt 1988] Staudt, B., C. Krueger and D. Garlan. *TransformGen: Automating the Maintenance of Structure-Oriented Environments.* Technical Report CMU-CS-88-186. Computer Science Department, Carnegie Mellon University. Nov. 1988.

[Straw et al. 1989] Straw, A., F. Mellender and S. Riegel. *Object Management in a Persistent Smalltalk System. Software–Practice and Experience,* 19, No. 8, Aug. 1989, pp. 719-737.

[Taylor et al. 1988] Taylor, R.N., R.W. Selby, M. Young, F.C. Belz, L.A. Clarke, J.C. Wileden, L. Osterweil and A.L. Wolf. *Foundations for the Arcadia Environment Architecture,* in *Proc. Symposium on Practical Software Development Environments.* Ed. P. Henderson. Association for Computing Machinery. Boston, MA: Nov. 1988, pp. 1-13.

[Tichy 1982] Tichy, W.F. *Adabase*: A Data Base for Ada Programs,* in *Proceedings of the Ada TEC Conference on Ada* . Association for Computing Machinery. Arlington, VA: Oct. 1982, pp. 57-65.

[Tichy 1986] Tichy, W.F. *Smart Recompilation. Transactions on Programming Languages and Systems,* 8, No. 3, July 1986, pp. 273-291.

[Ullman 1988] Ullman, J.D. *Principles of Database and Knowledge-Base Systems.* Potomac, Maryland: Computer Science Press, 1988. Vol. 1.

[Wileden et al. 1989] Wileden, J. C., A. L. Wolf, W. R. Rosenblatt and P. L. Tarr. *Specification Level Interoperability.* Technical Report. University of Massachusetts. 1989.

[Wileden et al. 1988] Wileden, J.C., A.L.: Fisher Wolf, C.D. and P.L. Tarr. *PGRAPHITE: An Experiment in Persistent Typed Object Management,* in *Proceedings of the Third Symposium on Software Development Environments.* Ed. P. Henderson. Association for Computing Machinery. Boston, MA: Nov. 1988, pp. 130-142.

[Wileden et al. 1990] Wileden, J.C., L.A. Clarke and A.L. Wolf. *A Comparative Evaluation of Object Definition Techniques for Large Prototype Systems. ACM Transactions on Programming Languages and Systems,* to appear (1990).

[Wolf et al. 1989] Wolf, A.L., L.A. Clarke and J.C. Wileden. *The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process. IEEE Transactions on Software Engineering,* 15, No. 3, Mar. 1989, pp. 250-263.