References

[1] F. P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[2] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[3] B. Lientz and E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, Reading MA, 1980.

[4] G.E. Krasner and S.T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, August/September 1988.

[5] G. E. Kaiser and D. Garlan. Melding software systems from reusable building blocks. *IEEE Software*, 4(4):17–24, July 1987.

[6] K.J. Sullivan and D. Notkin. Reconciling environment integration and component independence. In *ACM SIGSOFT '90: Fourth Symposium on Software Development Environments*, Irvine CA, December 1990. Special issue of *SIGSOFT Notices*.

shared data and event-driven tool integration. As explained in the breakout boxes, there are many instances of tool abstraction already in widespread use; these are summarized in the table.

The tool abstraction paradigm raises several interesting and difficult research problems. When multiple toolies react to the same event, ordering becomes an issue; there is also the potential problem of circularities among the dependencies implied by toolie events. When eventss are a programming-level paradigm, how can the independence of the toolies be retained while ensuring that circularities do not arise? Another problem is how to handle lazy vs. eager invocation of toolies. For applications concerned with meeting timing constraints, how can the time costs of indirect invocation be understood and managed? In general there are two approaches to handling these and other related questions. One is to constrain the domain and expressiveness of the tool abstraction language facilities.

We have discussed toolies only in the context of the kinds of data structures typically encapsulated in abstract data types. But tool abstraction might also support events other than operations on shared data structures. For example, it should be possible to attach toolie invocation to lack of data (when/where data is expected) and other exceptional conditions, timer interrupts and other signals, and events that are accessible only through polling of some external entity (e.g., sockets, sensors). Since software systems involving these kinds of events are typically quite complex, the tool abstraction paradigm should prove particularly fruitful.

Acknowledgments. We would like to thank Bill Griswold, Ralph London, Chip Maguire, Josephine Micallef, Harold Ossher, Steve Popovich, Kevin Sullivan, Michael van Biema, Travis Winfrey and Ursula Wolz for their detailed comments on drafts of this paper. Nancy Griffeth suggested the telecommunication example.

Table 1: Instances of Tool Abstraction

System	Shared Data	Toolies	Control Mechanism
Spreadsheets	2-D Matrix	Equations	Dependency Analysis
Structure-Oriented Environments	Abstract Syntax Tree	Action Routines	Operations
Structure-Oriented Environments	Abstract Syntax Tree	Attribute Equations	Dependency Analysis
Production Systems	Working Memory	Rules	Patterns and Rule Resolution
Object-Oriented Systems	Object Base	Methods	Active Values

decomposition; the difference is that his interfaces hide representations of data and other implementation decisions. Similarly, tool abstraction is not antithetical to abstract data types: indeed, the pool of data shared by a collection of toolies is itself an abstract data type. Moreover, it is reasonable to expect that a collection of toolies would be bundled up as an abstract data type that hides the details of that decomposition and enhancements would be made by adding toolies to the bundle; this is the approach taken in the Meld language [5].

What, then, is to be gained from all of this? The important point is that the abstraction paradigm or paradigms that drive a system's design should be chosen to reflect the kind of evolution expected of the system. If functional enhancements are expected to be the primary form of change (as they are in most large systems today), then tool abstraction represents a particularly attractive — and realizable — addition to existing approaches.

This has ramifications for three classes of people. *Language implementors* need to incorporate the principles of tool abstraction into new language designs and develop general implementation techniques. *Environment builders* need to develop facilities that support the design of systems based on tool abstraction, and analysis, testing and debugging aids that support the addition of new toolies to existing systems. Finally, *system designers* need to be aware of the tool abstraction paradigm and actively seek a system that supports

abstraction can be efficiently realized (although perhaps still in restricted domains).

Conclusion

There can be no absolute judgment about which criteria and integration mechanisms are best. A structuring technique that is appropriate in one circumstance may be inappropriate in others. System structures based on functional decomposition — the criterion criticized by Parnas — may be appropriate when physical data formats are unlikely to change, but new paths of processing existing data are expected as the system evolves: witness the success of Unix pipes. On the other hand, decomposition based strictly on data abstraction may be appropriate when data representations are likely to change, but not external interfaces. Finally, as we have argued, composition using tool abstraction is appropriate when a system evolves by enhancing the function supported by existing data structures.

Although our examples have been especially simple, functional enhancement does happen in complex situations. Consider the complexities that arise when enhancing basic telephone services with with services such as call forwarding and call waiting. For example, if call forwarding is turned on and the phone is in use, should the call waiting tone sound when an incoming call arrives or should the call be forwarded? In each case, data from the same sensors must be processed, the same billing data updated, and so forth. While the message queue and KWIC examples could reasonably be reimplemented from scratch when an enhancement is made, this is not feasible for most phone system features. Thus it becomes especially important for the implementation of new, optional phone services to be as independent as possible of the implementation of plain old telephone service.

Luckily, it is not necessary to choose one abstraction paradigm over the others. This is in part because there are no hard boundaries between these abstraction techniques. Parnas's decomposition of KWIC based on data abstraction appears very much like the functional

One dimension is the question of algorithmic complexity of the underlying system. Abstract data types, and their associated implementations, are generally designed to support a set of operations in an efficient manner. If the interface remains inviolate, it may no longer support new functions efficiently as the system evolves. In the example above, checking for duplicates using the simple enqueue/dequeue interface is an instance of this problem; the fixed interface makes what ought to be a constant-time operation into a linear-time operation (in terms of the number of invocations on the interface). In some cases, the complexity is the same but the constant can increase significantly; in practical systems, this may not be acceptable.

Another dimension is whether triggering mechanisms will slow down systems too much. There are at least two issues here. First, there is quite a bit of experience with triggering mechanisms in a wide variety of domains (again, refer to the breakout boxes for some examples). Many realistic systems have been built based on triggering mechanisms, so there seems to be no inherent hurdle to overcome. Second, any added costs here would be a matter at most of a constant factor, since the invocation of the operation would have to take place anyway: with triggers it's just a question of haggling about the price. This is in contrast to the algorithmic dimension, since restricted interfaces can slow programs down by greater than constant costs.

Language Issues

Parnas's initial paper presented the notion of data abstraction largely separate from language and implementation issues. In fact, data abstraction can be practiced without special language features, if the programmers are sufficiently disciplined. With tool abstraction, however, language and run-time support are necessary, since decreasing explicit modification of existing toolies eases incremental evolution. This is not a serious problem, however, since the systems described in the breakout boxes provide concrete evidence that tool

directly responsible for cooperatively managing their collective control flow. This may be simple at first, but it becomes increasingly complicated as enhancements are introduced.

Triggering can be implemented as an implicit, underlying mechanism or as an explicit, programmer-level mechanism. In attribute grammar-based structure-oriented environments, for instance, designers of specific environments are unaware of a triggering mechanism: they simply write the desired set of attribute equations. The same is true for spreadsheets. But the underlying implementation triggers incremental attribute evaluation schemes when attribute values are updated by the user. In production systems, the programmer is aware of how rules fire and must build systems that work based on these semantics.

When triggering is used as an implicit, underlying mechanism, many of the difficulties are managed by the underlying system. Circularity, for instance, can be a serious problem: toolies can indirectly invoke themselves producing an unbounded execution. Most attribute grammar and spreadsheet systems, however, check (either statically or dynamically) for such circularities, so the designer of an atribute grammar or a spreadsheet need not be overly concerned about circularities the resulting system. In action routine-based structure-oriented environments, however, the programmer must be aware of this kind of problem. The difficulties of trigger-based programming can be significant. The benefits, however, are also significant. There is is now a wealth of practical (although perhaps not systematically understood and documented) experience with trigger-based programming, from domains including structure-oriented environments, production systems, recent relational and object-oriented databases, object-oriented systems like the Smalltalk-80 Model-View-Controller [4], access-oriented programming as in Loops, etc.

Efficiency

We have addressed several dimensions of efficiency throughout the paper.

that one can easily build.

Tool Abstraction vs. Inheritance

Inheritance in object-oriented languages can be used to provide some aspects of tool abstraction. In particular, inheritance is an especially nice approach to extending abstract data types. In many cases, for instance, a subclass can then provide additional operations on an existing data type without modifying the base type.

Inheritance doesn't by itself provide all the features needed for tool abstraction. The most notable exception is events. Again, these can be added to object-oriented systems (as with the Smalltalk-80 Model-View-Controller [4]), but inheritance doesn't do the job by itself. Also, when inheritance is used to achieve sharing of code rather than sharing of behavior, the relationship to tool abstraction is even less clear. In particular, handling triggering effectively is especially difficult. Perhaps more fundamentally, most object-oriented systems do not encourage programming in the paradigm of tool abstraction, even when they provide many of the mechanisms that are needed.

Additionally, inheritance imposes a hierarchy on the data abstractions. Toolies, in contrast, are equals. In particular, one could define a system that uses a subset of existing toolies, picking and choosing from the desired functions. This is not straightforward to do with inheritance, where selecting a subclass implies selection of the properties of its superclasses.

Finally, inheritance can be viewed as a model for merging toolies together into a cohesive system. This is precisely the approach taken in Meld [5].

Events

Our examples rely on the use of an event (or trigger) mechanism. The basic reason is that triggers allow greater independence among toolies [6]. Without triggers, toolies would be

works for the **Omit** and **Include** toolies on the shared input buffer: by making the input and shift buffers explicit, separate toolies can be associated with each.

Discussion

Tool Abstraction vs. Pipes

Toolies are similar in intent to Unix pipes, which link together a preplanned sequence of small functional units (executable programs or shell scripts). Each functional unit in the pipeline takes as input the output produced by the previous unit; execution is triggered by the arrival of this input. Pipes are much more limited than toolies because pipes must be connected sequentially, while toolies share abstract data and are permitted to have much richer control interactions. A unit cannot react to operations on the data made by subsequent units in the pipe. Furthermore, in contrast to toolies, units connected by a pipe share only a single pre-defined data representation — a sequence of characters — with no capability for shared data definitions. When this is not a suitable representation for the internal processing of a particular unit, the unit must parse its input from and unparse its output to the standard character stream form. Each bundle of toolies can define their own data representation, with later enhancements perhaps adding components to the data structure.

These two restrictions do not pose a problem for all of the suggested enhancements to KWIC. For instance, pipes are a natural (although not particularly efficient) solution to omit/include on original/shifted lines: the input simply streams through a pipelined sequence of sorters and filters, and additional filters can always be inserted in the pipeline. In other situations with more complex interactions among functional units, pipes are inadequate. The tradeoff is not simple. The restrictions on pipes can be viewed as a way of managing complexity; but with the restrictions come a reduction in the kinds of systems

the alphabetized shifts, if desired, or the alphabetized buffer and the shift buffer could be equated to the same data structure. The sort could also be incremental, associating incremental insertions with the insertion of each shift into the shared shift buffer.

- The **Output** toolie provides a display scheme for printing the alphabetized shift buffer.

The top-level program invokes Input followed by Output. The actions of Input cause the Circular Shifter to execute, the actions of the shifter cause the Alphabetizer to execute, and Output simply accesses the sorted, shared buffer. An alternative approach would be for Alphabetizer to invoke an event when it is done sorting, and Output would be invoked and print the results automatically.

Potential Changes

Enhancements are accommodated more effectively and efficiently in this approach. Tool abstraction provides the ability to naturally define multiple enhancements independently of the existing code, while still producing programs that execute efficiently.

An **Omit** toolie can be associated with the insert operation (provided by the original **Input** toolie) on the shared shift buffer. If the first word of the shift being inserted is in the set of noise words, then the toolie aborts the insert (i.e., the **Omit** toolie is triggered by the initiation of the insert operation, not its termination). The code for the programs that insert into the shift buffer need not be changed at all. The cooperation between these toolies and the **Omit** toolie is triggered implicitly by operations invoked on the shared data rather than on explicit calls between operations. This eases evolution since multiple toolies can be triggered by the same operation without any changes at all to the triggering operation. The same approach holds for the **Include** toolie on the shift buffer. It also

The circular shifter is not concerned with the internals of the **Input** toolie, but only with its registered insert operation. (In practice, different systems handle registration is different ways; see the breakout boxes for some examples.) Alternatively, the shift creation code could be associated with the complete input buffer, rather than single lines, and therefore triggered when all input lines are inserted, when the **Input** toolie signals completion. Thus, in this example at least, tool abstraction is suited to defining incremental and batch computations. The input lines and the shifted lines are conceptually separate buffers. Changing between creating a new buffer and using the input buffer directly, or between using an explicit or an implicit representation of the shifts, can be done by redefining the **Circular Shifter** toolie. This illustrates how toolies can augment data as well as function.

- The **Alphabetizer** toolie associates its action with the shared shift buffer. The alphabetizer is triggered by the completion of the shifter's activities to sort the lines in the buffer. In the case of an implicit shift buffer, this would result in a coroutine interaction between the two toolies. Another shared buffer could be created to hold

data structures.

Design #2: Tool Abstraction

In the first design, the enhancements, while possible, are unnecessarily complicated. In particular, it is unfortunate that logically independent requirements are difficult to implement efficiently without intertwining logically independent implementations. For instance, deciding whether to include a shifted line should not have any affect on the implementation of the shift module, but that's the most effective implementation in practice. Tool abstraction, in contrast, allows each toolie to act as a (largely) independent entity that focuses on a single function, decreasing the complexity of successive enhancements.

With tool abstraction, the input data, the shifted data, and the alphabetized data are kept as shared (although still abstract) data structures. When a given toolie modifies the common data, other dependent toolies are invoked indirectly. This keeps each toolie clear of functionally unrelated code. The shared data are still abstract, since they are not "opened" completely. In particular, the physical representations are still hidden by the data abstraction mechanisms. The functions on these shared data entities are, however, factored into toolies. In contrast, traditional abstract data types combine these two concerns.

The second design is based on a collection of toolies that manipulate shared buffers representing sequences of lines. The following toolies define the basic operations of KWIC. Their interaction is illustrated in Figure 2.

- The **Input** toolie creates a new instance of a shared buffer. It reads lines from the input file, inserting each successive line into the shared line buffer.

- The **Circular Shifter** toolie creates another instance of a shared buffer to hold the shifted lines. It associates its action, in the sense of a *daemon* or *active data*, with the termination of the insert operation of the **Input** toolie so that as each line is originally inserted, the shifter is implicitly invoked to create the shifts for that line.

In this small example, the approach of modifying the shifter is not a serious problem. However, even including both the omission and inclusion enhancements in the shifter starts to make the module more complex. Rewriting it from scratch might produce a clean version, but it is not generally practical to rewrite whole modules each time an enhancement is made. So, in practice, the complexity of a module tends to explode as repeated enhancements are made by modifying a module or modules.

Two additional possible enhancements are closely related to the first two. In these, omission and inclusion are again provided, but on the original list of lines, rather than on the shifts. These enhancements, when combined with the first two, help produce KWIC indices that meet a wide range of needs. The same implementation tradeoffs arise. Filtering can be done after the line storage is initialized, again at added cost (although not as bad as in the shifter's case, since the insertion cost is linear). Or the line storage module itself can be modified, just as the shifter was.

One problem with modifying the shifter and line storage modules as suggested is that the decision as to whether to include a given line should not be the responsibility of those modules. This is not a serious problem for the simple enhancements we have discussed, but it becomes much more significant as other enhancements and modifications are introduced. For instance, what if a user wanted an enhancement where individual words could be included or not, as opposed to including or omitting shifts that start with these words? Implementing such changes efficiently in the line storage or the shifting modules would make the implementations confusing. Also, the ability to reuse modules, such as line storage, for storing other lists of words, such as the noise words, would be compromised, since orthogonal enhancements might be needed for each list. In any case, the focus of the modules on line storage or on shifting would fade, which is inappropriate since it compromises the separation of concerns. It should be possible to treat each enhancement as an independent unit, with the only interactions being through operations on the shared

associated line only if it does not start with one of the forbidden words. This approach is straightforward, but it is unnecessarily inefficient. In particular, all the shifts must be alphabetized, including those that will ultimately be omitted from the output. This added cost can be significant, as can be seen by looking at the KWIC index for Unix manual entries, which is based on a one-line header associated with each Unix command. If this index listed all the shifts, there would be about 5000 entries. But the actual index omits shifts starting with about 150 noise words: only about 1000 shifts actually appear when the noisy ones are filtered out. Sorting dominates the overall cost. In this case, the cost is $O(N \lg N)$ where N is the initial number of entries, so this decrease in entries to be sorted saves at least a factor of five in execution time. This cost is indicative of the performance penalties that may be required by restricting access to encapsulated data.

The second approach to implementing the omit version of KWIC is to modify the circular shifter. (The alphabetizing module could also be modified, but this approach is less attractive for similar concerns about performance.) As each call is made to the circular shifter to insert a new line, the line is checked against the set of prohibited words; if a match is found, the line is not inserted into the shifted list. The code to implement this approach is simple, and it keeps the structure of the implementation of the shifter straightforward. However, as we shall see, this kind of solution becomes complicated when further changes are considered.

A second possible enhancement is closely related to the first: given a set of words, include only those shifts that start with one of those words. This might be used to cull out a set of smaller KWIC indices, each related to a subtopic.

The same implementation approaches are available, with the same tradeoffs. Filtering is easy, leaving the existing modules unchanged, but inefficient. Or the "include" check can be incorporated into the shifter, in which case the code in the shifter becomes more complex and the question of how to handle other clients that use the shifter arises.

The top-level program first invokes the Input module, which stores the lines using the Line Storage module; the actual representation used by Line Storage is hidden from Input. Functions exported by the Circular Shifter module are then invoked; Circular Shifter retrieves the stored input using Line Storage, hiding decisions about data representation and algorithms from the top-level program. Then functions from the Alphabetizer module are used to sort the shifted data; Alphabetizer accesses the data through Circular Shifter while hiding the sorting algorithm from the top-level program. Finally, the Output module accesses the sorted list using Alphabetizer (and perhaps Circular Shifter). In contrast to the common functional decomposition, alternative control structures are quite easy to construct.

Potential Changes

Parnas's data decomposition is effective in handling alternative representations and implementations such as packed vs. unpacked characters, explicit vs. implicit representation of shifts, and monolithic vs. incremental alphabetization. However, Parnas's decomposition does not directly support other kinds of enhancements. This is not surprising, since Parnas was concerned primarily with situations in which the functional specifications of systems remained unchanged, although the implementations could vary. Looking at how some proposed enhancements might be supported in Parnas's decomposition of KWIC demonstrates some of the problems of evolution in systems modularized according to the principle of data abstraction.

Consider augmenting KWIC with the ability to omit shifts that start with one of a set of noise words, such as articles. Parnas's decomposition admits two approaches to making this modification.

The first approach is to include a simple filter in the output module (or add a small module that provides this filter). The filter checks the first word of each shift, printing the

Parnas describes the KWIC system as follows [2, p. 1054]:

> The KWIC index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

As Parnas explains, this is a small system and consequently none of the motivating issues actually arise. Similarly, it is hard to properly evaluate issues of evolution in a system of this system. However, it does allow us to point out the key issues and problems of tool abstraction, so, for the purposes of exposition, it is treated in his paper and ours as if it were a large project that presented realistic problems.

Design #1: Parnas's Decomposition

In Parnas's decomposition of the KWIC system the modules are designed to hide specific data representation and algorithms choices, so that these choices can later be changed.

- The **Line Storage** module implements a sequence of lines, with routines to create, access, and delete characters, words, and lines.
- The **Input** module reads and stores the original lines.
- The **Circular Shifter** module provides routines to access individual characters, words, and lines of the circular shifts of the stored lines.
- The **Alphabetizer** module provides routines to access shifted lines in alphabetical order.
- The **Output** module prints the circular shifts in alphabetical order.

enhancement, is invoked when the enqueue operation terminates successfully, augmenting the newly inserted message with a field that represents the current time. Maintaining correctness is relatively easy, since each new toolie interacts only with the message buffer representation and with any other toolie invoked by the same operation, not with other toolies associated with the message queue. In this case it would not be necessary to specify the order of invocation of these two toolies, since "remove duplicates" is invoked upon initiation of enqueue while "add timestamps" is invoked upon its termination.

An Example: KWIC Index Production System

As Parnas pointed out, the issue is not *whether* to modularize a system — since modularization is essential to the control of complexity — but rather the *criteria* one should use in decomposing a system design into modules. Parnas contrasted data abstraction with functional decomposition, showing that systems based on the former can better handle evolution for certain classes of change. We have taken this one step further. Tool abstraction reintroduces decomposition criteria according to function, but through a composition paradigm based on sharing abstract data structures (low-level abstract data types) among a collection of cooperating tools, together with an event-driven integration mechanism. A bundle of toolies represents a higher-level abstract data type, amenable to a wide range of enhancements. This approach contrasts with the strict use of data abstraction, and it leads to better support for the more common classes of functional enhancements.

Parnas used the KWIC Index Production System to compare two modularizations with respect to ease of evolution. To further illustrate the idea of tool abstraction, we do the same. The first design we present is Parnas's second decomposition, which uses data abstraction to decompose the system into modules. Our second design is based on the tool abstraction paradigm. We show how tool abstraction (along with data abstraction) supports evolutionary enhancements more effectively than does data abstraction *per se*.

be linear in the length of the queue. Another approach is to change the implementation of the queue module itself. The number of enqueue operations would drop to one, and no dequeue operations would be needed. However, this approach is unsatisfactory because the semantics (although not the syntax) of the enqueue operation must change. Thus, clients that want the original queue semantics (perhaps for some other use of queues) are not isolated from the change. Using data abstraction, this kind of problem must be handled by creating distinct abstract interfaces, each with different semantics.

Both approaches are less attractive in the face of multiple enhancements. Consider a second enhancement that adds a timestamp to each message; this necessarily interacts with the prohibition against duplicate messages. In particular, both enhancements must modify the implementation even though they may be conceptually independent. Further, the clients must usually be modified to gain any advantages ¿from the timestamp enhancement. Again, this is further complicated when different clients desire different semantics: some clients might prefer to queue the duplicate with the earliest timestamp while others prefer that with the latest timestamp.

Although this example is especially simple, it illustrates some of the tensions that arise when systems evolve. The use of tool abstraction shows how the problems of evolution can be reduced.

The shared data structure is the message buffer. The basic enqueue and dequeue operations are provided by one toolie. To handle the first enhancement, a "remove duplicates" toolie is defined; after the initiation of each enqueue operation, this toolie is invoked to compare the message about to be inserted with all the other messages, aborting the enqueue operation if the message is already in the queue. Because it has direct access to the message buffer, the toolie can be implemented reasonably efficiently. (If needed, the toolie can maintain an auxiliary representation of the buffer, keyed by whatever component of a message is checked for duplication.) The "add timestamps" toolie, which defines the second

approach.

Tool abstraction complements, rather than supplants, data abstraction; in particular, data abstraction allows design decisions about the representation of data structures to change easily, while tool abstraction is used to allow the function of the system to change easily.

A Simple Example

Consider a small message system centered around a queue module that exports enqueue and dequeue operations only. Using abstract data types, how would we enhance the system so that a duplicate message is never added to the queue? One approach is to modify the clients of the queue. This is unsatisfactory because to detect duplicates using the original interface, the client would have to dequeue every message already on the queue, compare it to the new message to be enqueued, and then re-enqueue the original messages plus perhaps the new one. Even though this activity could be encapsulated in a new client, the performance penalty is severe. Specifically, the number of enqueue/enqueue operations executed would

enhanced incrementally and modifications to be developed independently, even when the changes cannot be hidden using traditional data abstraction techniques. Several existing kinds of systems approximate these objectives. As one example, spreadsheets are often enhanced by adding new equations that use the values in data cells to interact indirectly with existing equations. As another example, production systems—a popular implementation paradigm for expert systems—consist of a collection of independent pattern-action pairs (called rules) that fire when the patterns match values a shared database (called working memory). In principle, a production system can be enhanced by adding new rules that match and manipulate the working memory.

The diverse set of systems that are structured in this style can be seen as instances of what we call the *tool abstraction* paradigm. That is, despite their differences, each system of this sort has a common structure that encourages and eases incremental enhancement of a system's function, just as data abstraction encourages and eases the changing of design decisions about data representation. Rather than detracting from the central idea of tool abstraction by introducing a new tool abstraction mechanism, we use existing approaches to describe and argue the benefits of tool abstraction. The breakout boxes describe several of these approaches and how they realize tool abstraction.

Systems that support tool abstraction are structured as a pool of abstract data structures, shared by a collection of cooperating *toolies*, where each toolie provides a piece of the overall data and function of the system. When one toolie updates the shared data, other toolies must be notified; otherwise, cooperating but independent toolies may not execute, and the overall system function may be compromised. This architecture is illustrated in Figure 1.

At this level, tool abstraction is similar to trigger-based database management systems that provide access to shared data through a common set of schemas. As discussed in a later section, each system handles notification differently, although most use an event-based

Introduction

The management of complexity and support for evolution are two fundamental problems with large scale software systems [1]. Modularization has long been accepted as the basic approach to managing complexity. However, nearly two decades ago David Parnas observed that not all modularizations are equally good at handling evolution [2].

Data abstraction is one popular and important style of modularization. In this style, an abstract data type is defined by an explicit interface that specifies operations on instances of the type while deferring design decisions about the representation of concrete data structures and the implementation of algorithms on those data structures. These concrete decisions can be changed without modifying the clients, which are written in terms of the stable interface, easing some kinds of evolution.

Enhancing a system's function is common, typically accounting for about 60% of the maintenance costs in a large system's lifecycle, and hence roughly 40% of the total software lifecycle costs [3]. If the interfaces to abstract data types are kept unchanged, which is needed to protect clients from evolutionary changes, then enhancements must often be constructed in terms of existing abstract data types. This restriction can lead to two problems: first, the desired function may not be computable from the existing interfaces; or second, implementing the function in terms of the existing interfaces may be unacceptably inefficient.

Thus, modifying the abstract interface itself may be the only or the most effective way to make the enhancement. Since changing the abstract interface implies that the concrete implementation must be understood and changed, the complexity of making a change is often increased. While a single such change to an existing data abstraction may not be a serious problem, as the number of enhancements continues to grow, so does the complexity of the interactions between those enhancements.

What is needed is an approach to handling changes that permits the system to be