Formal Manipulation of Modular Software Systems

Robert L. Nord Peter Lee William L. Scherlis

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890

Abstract

We present a formally based method for systematically integrating software components. This is accomplished through the adjustment and mediation of abstract systems interfaces and their underlying data representations. The method provides the software designer with the ability to delay or revise design decisions in cases when it is difficult to reach an *a priori* agreement on interfaces and/or data representations.

A moderate-scale example, drawn from the development of a simple interactive text editor, is provided to demonstrate the application of these techniques. The text buffer in an editor must support a variety of operations. These fall into groups determined by the most natural and efficient data representations that support the individual operations. We demonstrate how such data representations can be combined using formal program manipulation methods to obtain an efficient composite representation that supports all of the operations.

This approach can provide meaningful support for later adaptation. Should a new editor operation be added at a later time, the initial representations can be reused to support another combination step that obtains a new composite representation that works for all of the operations including the new one.

This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

1 Introduction

Key to the management of larger-scale software systems is the organization of interfaces among system components. For many large systems, a principal source of risk is the set of decisions concerning the placement of these interfaces — how the components are to be organized into a systems architecture. Language features for modularity, including the various advanced type systems, provide a means for component structure to be made more explicit, thus facilitating management of systems interfaces.

We suggest that formal methods can be applied to support the development and evolution of larger-scale systems through the formal manipulation of the interfaces and components. As a system architecture matures and evolves, interfaces and components may need to be adjusted in various ways, by moving or shifting computations across interfaces, by introducing new interfaces to create new components, by combining similar interfaces to merge components, and so on. Indeed, the architecture of large systems is rarely determined fully in advance, and, in any case, evolves rapidly as development experience is gained. Formal methods can provide a basis for the creation of software tools that can support this kind of iterative refinement. In this paper, we address the issue of how formal program manipulation techniques can be applied to support iterative refinement while preserving those program meanings that need to be preserved.

Consider, for example, the development of an interactive display editor. A key subproblem is the definition of operations on the data structure for the text buffer. There are many possible representations for buffers, for example a sequence of characters, a sequence of lines, and so on, and for each operation, one representation may be more natural or appropriate than another. Rather than having to decide in advance on some compromise, would it not be preferable to collect into separate components the sets of individual editing

Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development. Napa, California, May 9-11, 1990.

operations that agreed on associated "natural" representations for straightforward implementation? This is a very natural step to take as a paper exercise in the initial design of a system, before arriving at the ultimate data structure definition that must work for all the operations. The approach we suggest involves taking individual components, each using its own "natural representation," and aggregate them into a single composite implementation. This requires finding a way to mediate the interactions among the components.

Program transformation techniques can be of assistance in accomplishing this. Before discussing this however, we first consider the strategies that are currently available to the software-system designer.

2 Formal Methods in Software Development

Modern programming languages such as Ada [3], Clu [15], and Modula-2 [23] have data abstraction and encapsulation constructs called packages, clusters, or modules that enable one to define and enforce the boundaries separating the components of a software system. Modularity facilitates reuse and analysis and, when properly structured (either by design or through evolution), isolates and localizes the revisions that occur as a system is maintained, adapted, and reused. In this paper, we refer to these data abstraction constructs as *modules*. Modules can be viewed as a kind of (usually complex) data type definition. Like data type definitions, modules consist of what we call an *abstract interface*, that is, the exported types and signatures of the operations; the underlying *representations* for the data objects; and the *implementations* of the operations. We refer to an abstract interface and its associated data representations collectively as a *module interface*. (We are motivated to make this definition of module interface by the fact that both the abstract interface and the data representations affect the interactions among data objects in a system.)

Unfortunately, the integration of such data types in a larger-scale system is difficult; types that interact must agree not only on the abstract interfaces, but also on data representations in the cases where they share data. Also, as a software system evolves, the need to adapt existing interfaces can arise. Thus, this problem of integration persists for as long as the system is maintained.

The Existing Choices in Software Development. Confronted with the problem of integrating interfaces in larger-scale systems, the software-system designer has the following choices:

1. Make an *a priori* correct choice of abstract interface and data representation definitions that will suffice for all anticipated needs.

   Unfortunately, common data representations may be difficult to design *a priori*, especially when there is not much experience in the particular application domain. Once built, systems also evolve as users desire additional functionality which may not be anticipated. Clearly all needs for novel application domains and evolving systems cannot be anticipated. Adapting components is usually difficult once design decisions are made and, indeed, the cost of implementing change often becomes unmanageable. These problems have led software designers toward iterative and evolutionary models of development [4], but little advice is given on how to get from one stage to the next.

2. Introduce functions for translating between representations in the situations where the abstract interfaces agree but the data representations do not.

   Separately designed modules that share data may be used together by writing translation functions that convert from one module's representation for data objects to the other's. Unfortunately, efficiency is lost in the overhead of mapping back and forth among modules.

3. Use a very-high-level language with built-in high-level types.

   In this case data representations are not explicitly defined. Instead, design decisions regarding data representations are left to a compiler. Unfortunately, the performance of the implementation and expressiveness of the programming language are limited by the existing compilation technology. If a designer wants to develop a system using rich abstractions that will have exacting performance requirements, then it seems that the designer must be involved in defining data representations.

4. Adapt or refine the abstract interfaces of existing modules by defining new modules as extensions of the existing ones.

   For example, object-oriented techniques can be used to define new types (and hence abstract interfaces) in terms of existing ones [16]. Objects having the new type will share meaning with objects of the the existing type, typically by inheriting its operations and adding something more. The new objects will also share implementation by directly reusing the *code* for the existing objects. Unfortunately there is no way to specialize that implementation in the context of the new type in order to obtain better performance.

The Role of Formal Methods. Each approach addresses the problem of integration with varying degrees of success. We are interested in investigating how program transformation methods might be used to complement or enhance them. In particular, we are exploring the use of transformation-based techniques to (1) provide a source of guidance on how to get from one stage to the next in evolutionary models of development, (2) alleviate the overhead of translation functions through program manipulation, (3) optimize programs in very-high-level languages, and (4) specialize implementations to obtain better performance in object-oriented programs that reuse code through inheritance. The goal of our research is to determine how transformation techniques might be developed to aid the process of building and managing complex types (or modules).

Criteria to help focus our evaluation of the utility of these techniques include:

- *Scalability.* What techniques are available for constructing larger programs? This is a major source of motivation for this investigation of transformations and module interfaces.
- *Expressiveness.* To what degree are the conceptual properties of the problem reflected in the syntax of the language? It is conventional wisdom that module facilities enable more explicit representation of systems architecture, and, through information hiding, enable components to be designed and developed separately. The challenge is to develop module mechanisms and formal methods approaches that can exploit modularity, and to develop formal methods approaches that support aggregation and integration of components.
- *Appropriateness of Representations.* How do data representations reflect the requirements of each component? As a system evolves, compromises are inevitably made to data representations in order to meet diverse needs; often expedient solutions are developed from which a later retreat is required.
- *Interface Agreement.* When must agreement on interfaces in the design of software be reached? Delaying design decisions when *a priori* agreement can not be reached may make the design process easier initially but additional work is usually required to integrate the components later.
- *Adaptability.* How easy is it to modify existing interfaces and incorporate new components into the system?
- *Reliability.* How can higher assurance of correctness be provided for larger systems, or, rather, for aspects of behavior of larger systems?
- *Performance.* What choices are available to the designer for improving the efficiency of a program? These might include techniques that affect the frequency of execution of parts of the program and how readily information is made available.
- *Automation.* How can the process of producing efficient programs from high-level programs be mechanized? The main emphasis here should be on achieving productive interactions of automated systems with developers and maintainers.

We will return to these criteria after presenting our approach, in order to compare it with other approaches to formalized software development.

Our Approach. We are developing program transformation methods to integrate module interfaces yielding efficient implementations. With these techniques, complex data type definitions can start as a collection of separate modules. Then, *translation functions* are used to reach agreement on data representations, and *module extensions* for defining new interfaces are used to reach agreement on abstract interfaces. The initial interfaces are then integrated by using an extended form of data-type transformations. Thus, a consistent and efficient implementation, but possibly with complex composite interfaces, is obtained. The techniques may facilitate the application of program transformation to larger-scale programs.

3 Deriving and Manipulating Module Interfaces

In order to demonstrate our techniques for integrating module interfaces by transformations, we will show part of the development of a simple interactive text editor. There is space enough only for a few highlights of the complete derivation. (More details may be found in a separate report [18].) Our editor is designed as a collection of separate modules. The collection of modules is then integrated and adapted by deriving module interfaces to achieve an executable prototype. This prepares the way to introduce efficiency transformations later in the derivation process. The entire process is depicted in Figure 1, which will be referred to in the following example as the steps are elaborated. But before we go into the details of our example, we must first say a few words about the previous work on data transformations that forms the basis for our techniques.

Data Transformations. The use of data abstraction in programming suggests there is value in examining transformation techniques on modules. Early methods focused on the relationship between abstract programs and their implementations. Hoare [12] presents

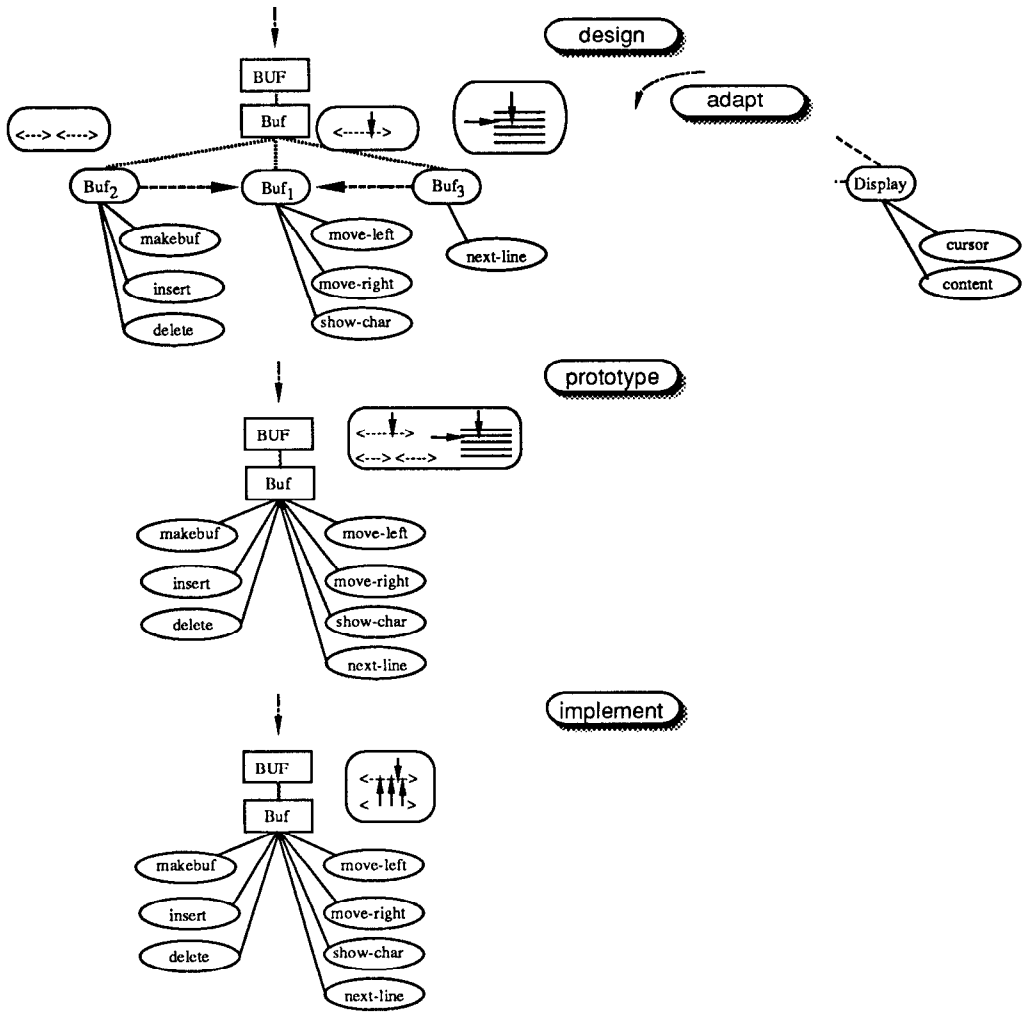


Figure 1: Deriving a Buffer

a method for proving the correctness of a data representation for an abstract program. This approach has been adopted by VDM [2]. An alternative approach is to derive the concrete representation using program transformation [5, 8] rather than invent the concrete representation and then prove it correct. Darlington [7] shows how a concrete program can be derived from an abstract program using program transformations that ensure that the implementation is correct. Wile [22] develops this idea further by considering the interrelationships along data paths in programs and outlining a set of informally described operations on data types. These include operations for delaying or advancing computation and operations for changing type signatures based on the "theory operations" of Burstall and Goguen [6]. Jørring and Scherlis [14, 21] develop and generalize these ideas to obtain a framework that permits programmers to take general purpose abstract type definitions and, using type transformations, obtain types tailored to the application.

We now present our example derivation.

Notation. In the following examples, the `typewriter` font is used for data types, lower-case greek letters for type variables, sans-serif font for functions, and *italics* for variables. The product type constructor "×" binds more tightly than the function type constructor "→". Function definition is denoted by "⇐". Data type definitions are represented as modules using an extended form of Standard ML [17]. The extensions and additional notation will be described as they are introduced in the examples.

An Editor Buffer. The abstract interface for an editor buffer is defined as a signature containing, for

the purposes of our example, seven buffer operations: makebuf, delete, insert, move-left, move-right, show-char, and next-line.

```
Signature BUF = sig
  type buf
  makebuf :   buf
  delete :    buf → buf
  insert :    ch × buf → buf
  move-left :   buf → buf
  move-right :   buf → buf
  show-char :   buf → ch
  next-line :   buf → buf
  end
```

Our intended goal is to arrive at one data representation that implements all of these operations efficiently. Since designing an efficient data representation that satisfies all of the operations may be difficult, we will chose to implement subsets efficiently, and then try to integrate them.

Program Composition. An appropriate model to represent a buffer on which move operations can be easily defined is a sequence of characters with an explicit index for the point where editing takes place. This point is marked by the cursor. The cursor is moved left by decrementing the index, "p-", and moved right by incrementing the index, "p+". The character at the cursor is shown by looking up the character in the text to the left of the index, "$t[p\text{-}]$". We call this a *component*, rather than a Standard ML "structure" since it implements only a subset of the buffer signature. Abstraction boundaries are maintained using a quasi-equational notation that is similar to abstraction and clausal definitions in Standard ML. (Constraints on the component, for example, that the cursor remains within the text, have been omitted to simplify the presentation.)

```
Component Buf1 : BUF = struct
  type buf = Buf of (int × ch*)
  move-left(Buf(p, t))   ⇐   Buf(p-, t)
  move-right(Buf(p, t))  ⇐   Buf(p+, t)
  show-char(Buf(p, t))   ⇐   t[p-]
  end
```

This provides simple and natural definitions for the three operations shown. To include insertion and deletion of characters in this $\mathtt{Buf}_1$ representation, on the other hand, requires a bit of manipulation of subsequences within the text. A more appropriate representation for these new operations (from the view of conceptual simplicity) might be a pair of sequences, representing the characters to the left and to the right of the point of editing. The index for the point of editing is implicit. A character is deleted by removing the last element from the left sequence, "front(l)." A character is inserted by appending it to the left sequence, "$l \circ [c]$".

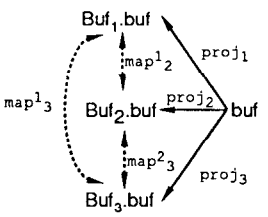
```
Component Buf2 : BUF = struct
  type buf = Buf of (ch* × ch*)
  makebuf   ⇐   Buf([], [])
  delete(Buf(l, r))   ⇐   Buf(front(l), r)
  insert(c, Buf(l, r))   ⇐   Buf(l ∘ [c], r)
  end
```

The next-line operation moves the cursor to the following line with the character position in the line remaining the same. This is difficult to do using either of the previous representations, since it would require searching for newlines and computing the distance between the point of editing and the preceeding newline. For this new operation, a new component, $\mathtt{Buf}_3$, is introduced where the text is a sequence of lines (where a line is a sequence of characters not containing a newline) and the point of editing is a line and character position. Now the cursor is moved to the next line by incrementing the line position by one, "lp+".

```
Component Buf3 : BUF = struct
  type buf = Buf of ((int × int) × line*)
    type line = (ch - 'nl')*
  next-line(Buf(⟨lp, cp⟩, ts))   ⇐   Buf(⟨lp+, cp⟩, ts)
  end
```

The newlines are implicit, giving a compact representation; however, an alternative representation could be used that keeps a newline character at the end of each line. The choice is up to the designer.

Designing Interfaces. Collectively the components implement all of the operations of the signature for BUF. However, an agreement among representations of the components must be reached to link them together so that all the operations can be executed on a single buffer. That is, when the move-right operation is executed, for example, not only is the $\mathtt{Buf}_1$ component updated, but the $\mathtt{Buf}_2$ and $\mathtt{Buf}_3$ components must be updated as well. The aggregate buffer can thus be defined in terms of the components, along with functions that translate among the components. Each component is a projection of the buffer and is made consistent with each other component via the translation functions.



axiom $\mathsf{proj}_1(\mathsf{move\text{-}right}(b)) = \mathsf{Buf}_1.\mathsf{move\text{-}right}(\mathsf{proj}_1(b))$

axiom $\mathsf{map}_1^2(\mathsf{proj}_1(b), \mathsf{proj}_2(b)) \Longrightarrow \mathsf{map}_1^2(\mathsf{proj}_1(\mathsf{move\text{-}right}(b)),\ \mathsf{proj}_2(\mathsf{move\text{-}right}(b)))$

axiom $\mathsf{map}_3^1(\mathsf{proj}_3(b), \mathsf{proj}_1(b)) \Longrightarrow \mathsf{map}_3^1(\mathsf{proj}_3(\mathsf{move\text{-}right}(b)),\ \mathsf{proj}_1(\mathsf{move\text{-}right}(b)))$

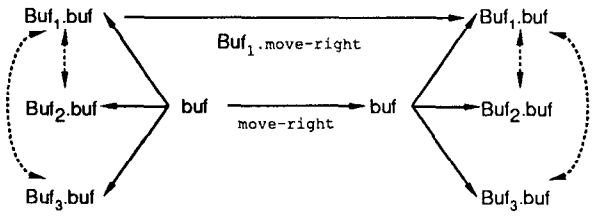
Figure 2: Buffer Specification

Structure Buf : BUF = **struct**
  structure Buf_1, Buf_2, Buf_3
  type buf = Buf **of** (int × ch* × ch* × ch* × (int × int) × line*)

  $\mathsf{makebuf}_{1\times 2} \Leftarrow \mathsf{span}_a(\mathsf{Buf}_2.\mathsf{makebuf})$
  $\mathsf{unspan}_b(\mathsf{makebuf}) \Leftarrow \mathsf{makebuf}_{1\times 2}$
  ...
  $\mathsf{unspan}_c(\mathsf{move\text{-}right}(\mathsf{Buf}(p,t,l,r,\langle lp,cp\rangle,ts))) \Leftarrow$
    $\mathsf{Buf}_1.\mathsf{move\text{-}right}(\mathsf{unspan}_c(\mathsf{Buf}(p,t,l,r,\langle lp,cp\rangle,ts)))$
  $\mathsf{show\text{-}char}(\mathsf{Buf}(p,t,l,r,\langle lp,cp\rangle,ts)) \Leftarrow \mathsf{Buf}_1.\mathsf{show\text{-}char}(\mathsf{unspan}_c(\mathsf{Buf}(p,t,l,r,\langle lp,cp\rangle,ts)))$
  $\mathsf{next\text{-}line}_{1\times 3}(\mathsf{span}_d(\mathsf{Buf}_3.\mathsf{Buf}(\langle lp,cp\rangle,ts))) \Leftarrow \mathsf{span}_d(\mathsf{Buf}_3.\mathsf{next\text{-}line}(\mathsf{Buf}_3.\mathsf{Buf}(\langle lp,cp\rangle,ts)))$
  $\mathsf{unspan}_e(\mathsf{next\text{-}line}(\mathsf{Buf}(p,t,l,r,\langle lp,cp\rangle,ts))) \Leftarrow \mathsf{next\text{-}line}_{1\times 3}(\mathsf{unspan}_e(\mathsf{Buf}(p,t,l,r,\langle lp,cp\rangle,ts)))$
  $\mathsf{map}_{2\to 1}(\mathsf{Buf}_2.\mathsf{Buf}(l,r)) \Leftarrow \mathsf{Buf}_1.\mathsf{Buf}(\#l,\ l \circ r)$
  $\mathsf{map}_{3\to 1}(\mathsf{Buf}_3.\mathsf{Buf}(\langle lp,cp\rangle,ts)) \Leftarrow \mathsf{Buf}_1.\mathsf{Buf}(\#(\mathsf{lines\text{-}to\text{-}chars}(ts[..lp-])) + cp,\ \mathsf{lines\text{-}to\text{-}chars}(ts))$
    where $\mathsf{lines\text{-}to\text{-}chars}(s) =$ **if** $\mathsf{null}(s)$ **then** [] **else** $[\mathsf{hd}(s)] \circ$ ['nl'] $\circ\ \mathsf{lines\text{-}to\text{-}chars}(\mathsf{tl}(s))$
  $\mathsf{span}_a(\mathsf{Buf}_2.\mathsf{Buf}(l,r)) \Leftarrow \mathsf{Buf}_{1\times 2}(p,\ t,\ l,\ r)$
    where $\mathsf{Buf}_1.\mathsf{Buf}(p,t) = \mathsf{map}_{2\to 1}(\mathsf{Buf}_2.\mathsf{Buf}(l,r))$
  $\mathsf{unspan}_b(\mathsf{Buf}(p,t,l,r,\langle lp,cp\rangle,ts)) \Leftarrow \mathsf{Buf}_{1\times 2}(\{\ p \mid p_3\ \},\ \{\ t \mid t_3\ \},\ l,\ r)$
    where $\mathsf{Buf}_1.\mathsf{Buf}(p_3,t_3) = \mathsf{map}_{3\to 1}(\mathsf{Buf}_3.\mathsf{Buf}(\langle lp,cp\rangle,ts))$
  ...
  end

Figure 3: Buffer Definition

The effect of an operation on the buffer is defined in terms of the component in which it was defined. The new operation updates the buffer correctly and all other components are kept consistent as expressed in the following commutative diagram.



The relationships depicted in this diagram can be specified via axioms on the operations in the various components. Such axioms can be written along with the signature of the data type. The annotated signature then constitutes a specification of the integration of the components. We show the axioms for move-right in Figure 2.

These axioms suggest that an extended form of data transformations might be useful in the design and implementation of the buffer data type (Figure 1, design step). The basic principle of data transformations is as follows: Given a program f on a domain D and a function that maps elements of the domain D' to elements of the domain D, we can define the following "expression procedure" [20]:

$$\mathsf{Abs}(\mathsf{f}'(d)) \Leftarrow \mathsf{f}(\mathsf{Abs}(d))$$

Then, syntactic transformations can be used to obtain an executable definition for the program f′ on the domain D'.

We can extend these methods to data aggregates as well. When we are given a collection of components for the buffer, perhaps the easiest way to integrate them is to define the buffer representation as the product of the component representations. The buffer definition shown in Figure 3 uses this approach. (As is data transforma-

```
Structure Buf_proto : BUF = struct
  type buf = Buf of (int × ch* × ch* × ch* × (int × int) × line*)
  makebuf   ⇐   Buf(0, [], [], [], ⟨0,0⟩, [])
  ...
  move-right(Buf(p, t, l, r, ⟨lp, cp⟩, ts))   ⇐
    let lp', cp' = if (cp = #ts[lp]) then lp+, 0 else lp, cp+ in
      Buf(p+, t, l ∘ [hd(r)], tl(r), ⟨lp', cp'⟩, ts)
  show-char(Buf(p, t, l, r, ⟨lp, cp⟩, ts))   ⇐
    { t[p-] | last(l) | (if (cp = 0) then 'nl' else ts[lp][cp-]) }
  next-line(Buf(p, t, l, r, ⟨lp, cp⟩, ts))   ⇐
    let d = (nlpos(ts))[lp] − (nlpos(ts))[lp-] in
      Buf(p + d, t, l ∘ r[..d], r[d+..], ⟨lp+, cp⟩, ts)
  end
```

Figure 4: Buffer Prototype

tions, these operations are defined by expression procedures which are more general than ML patterns since an expression can appear on the left hand side of the function definition.) The operations are defined in terms of the aggregate buffer representation, or some portion of this representation. To map between the components and the various aggregates, "span" and "unspan" functions are used—this also has the effect of putting things into a form suitable for data transformation.

In the definition, **Buf** implements the signature **BUF** using the components $\mathtt{Buf}_1$, $\mathtt{Buf}_2$, and $\mathtt{Buf}_3$. A representative sample of operations is shown. Defining the makebuf operation for the aggregate requires two stages because the $\mathtt{Buf}_2$ component, in which it is defined, is not directly connected to all the other components. It is connected directly to $\mathtt{Buf}_1$ via a translation function, but is connected indirectly to $\mathtt{Buf}_3$. In the first stage, an intermediate definition of makebuf is defined on an intermediate aggregate (the product of $\mathtt{Buf}_1$ and $\mathtt{Buf}_2$). In the second stage, the final operation on the aggregate buffer is defined by merging this intermediate definition with the $\mathtt{Buf}_3$ component. Since the $\mathtt{Buf}_1$ component is directly connected to all other components, new implementations for the operations defined in this component (*e.g.*, move-right and show-char) can be defined in a single step.

The components are kept consistent through the translation functions $\mathsf{map}_{2\to1}$ and $\mathsf{map}_{3\to1}$. It is not necessary that all translations among components be given; it is sufficient that the components are connected, possibly through some number of intermediate components. Component $\mathtt{Buf}_2$ is mapped into component $\mathtt{Buf}_1$ by making the point of editing explicit (which is the number of characters to the left of the point, "#*l*"), and by appending the left and right sequence of characters together. Component $\mathtt{Buf}_3$ is mapped into component $\mathtt{Buf}_1$ by converting the line and character indices into a character index and by converting the sequence of lines into a sequence of characters. The auxiliary function lines-to-chars takes a sequence of lines, adds a newline to the end of each one and appends them to make a sequence of characters. (The notation "$s[..i]$" denotes the subsequence of s from the beginning of s to i inclusive.) The span functions can be easily defined in terms of the map equations. A value that can be computed in more than one way is denoted "$\{c_1|c_2|c_3\}$." Multiple alternative ways to compute a value are maintained to ensure consistency among the components.

Deriving Interfaces. Next a prototype (Figure 4) is derived (Figure 1, prototype step) where the expression procedures defining the buffer operations are transformed into functional definitions. For brevity, the steps have been omitted. As with other data-type transformations, they consist of a number of purely mechanical steps and a few insight steps that require input from the designer. It is not actually necessary to derive span functions that are computable. Instead, the transformation process makes use of them in syntactic manipulations to obtain computable functions for the buffer operations.

In the prototype the data representation is simply the product of the data representations of the components. All components are updated simultaneously.

The makebuf operation generates each component representation. The move-right operation increments the index appropriately for each component. (The functions hd and tl return the first element and the rest of a sequence.) The show-char operation returns the character at the cursor position of the buffer. The value may be produced from any of the three representations;

```
Structure Buf_impl : BUF = struct
  type buf = Buf of ((int × ch*) × (int × int*))
  makebuf   ⇐   Buf(0, [], 0, [])
  ...
  move-right(Buf(p, t, i, nl))   ⇐   Buf(p+, t, (if nlp(t[p]) then i+ else i), nl)
  show-char(Buf(p, t, i, nl))   ⇐   t[p-]
  next-line(Buf(p, t, i, nl))   ⇐   Buf(p + (nl[i] − nl[i-]), t, i+, nl)
  end
```

Figure 5: Buffer Implementation

these three alternatives are denoted "$\{c_1|c_2|c_3\}$." This is the only extension to Standard ML in the prototype. The extension could easily be implemented by selecting the first alternative so that the prototype could be executed. Multiple alternative ways to compute a value are kept in order to avoid losing information that may be useful in later transformation or analysis steps. (The function last returns the last element of a sequence.) In the next-line operation, the positions of the surrounding newlines are used to advance to the next line for the $\texttt{Buf}_1$ component. (The function nlpos takes a sequence of lines and returns a sequence of newline positions.) Note that the character index for the $\texttt{Buf}_1$ component has been transformed to use information from the $\texttt{Buf}_3$ component, where it is easier to compute newline information. The representation of each component is available to update any other component representation. How they are used provides the motivation for the final representation that follows.

Shifting Computation. A specialized implementation (Figure 5) is derived using data-type transformation techniques to obtain a representation that caches newline positions (Figure 1, implement step). For brevity, the steps again have been omitted. Performance is improved by eliminating the computation for the $\texttt{Buf}_2$ component, and computing newline information directly rather than maintaining a sequence of lines and mapping it into newline positions when needed. The former is an instance of *releasing* components from the data type, while the latter is an instance of *shifting* computation from access to creation time, techniques that are described in [13, 21].

The new specialized representation is a sequence of characters with an index for the cursor and a sequence of newline positions with an index tracking which line contains the cursor. The makebuf operation now generates an empty buffer and newline cache. The move-right operation updates the newline index when crossing over a line. (The predicate nlp returns true when its argument is a newline.) The show-char operation is one of the three choices. The next-line operation uses the newline cache to move more efficiently.

Adapting Interfaces. The editor buffer can now be adapted further, for example by including a display. A display will need to know the cursor position and the content of the buffer; operations that the editor buffer module does not currently support. An agreement between the abstract interfaces of the editor buffer and display can be reached by extending the editor buffer module to include these additional two operations. One way to do this is to add a new component consisting of the operations required by display and re-derive the module interfaces (Figure 1, adapt step).

4 Conclusions

Our research extends data transformations by introducing transformation techniques for the module and interface design process. Once a complex type definition is defined as a collection of components, these transformation techniques provide a systematic way to integrate them. The components are first aggregated to produce a "canonical" data representation, which is a straightforward combination of component representations. Additionally, the representation-translation functions are incorporated into the operations. Then, the components are coalesced by using transformations to specialize the components in the context of the aggregate data structure and eliminate redundant information or computation, thereby producing an efficient aggregate implementation.

Our experience with the derivation of an editor buffer has given us some encouragement that formal program manipulation techniques might be applicable to larger software systems. The encouragement derives not from the scale of the buffer example, which is modest, but from the complexity of the interfaces and the means by which they were obtained. Scalability, nonetheless, must be demonstrated by scaling up, and therefore a larger derivation is underway that will cover a larger,

more realistic set of operations and components, and also will include display manipulation.

Automated assistance would be useful for managing the derivation, especially for the larger problems. In this case, the bulk of the transformation steps could be automatically applied, with the user left only with the task of selecting the appropriate strategies and providing the "insight steps" in the derivation. Still, discovery of the insight steps will likely turn out to be a significant bottleneck. We cannot expect to eliminate this entirely, but can work toward isolating the steps into manageable local pieces each of which requires a minimal amount of context.

The Existing Choices in Software Development — Revisited. The approach we have demonstrated addresses the problems of integrating module interfaces (raised earlier in this paper) in the following ways:

1. Rather than requiring *a priori* agreement on data representations, complex data types are defined as a collection of separate modules that are systematically merged using formal methods to derive the module interfaces and efficient representations.

2. Rather than mediating representations through intermediate translation functions, new module interfaces can be derived that interact directly.

3. Rather than leaving data design decisions to a compiler when using very-high-level languages, the software designer is involved in defining and organizing module interfaces.

4. Rather than requiring *a priori* agreement on abstract interfaces, new types may be defined as extensions of existing ones using module extensions (*e.g.*, object-oriented techniques using inheritance) and new module interfaces can be derived.

The Role of Formal Methods — Revisited. There are a number of other approaches to formalized software development. In this section, we indicate the different choices and tradeoffs some of them make with respect to the criteria that are suggested in Section 2: scalability, expressiveness, appropriateness of representations, interface agreement, adaptability, reliability, performance, and automation.

There are a number of formal frameworks for developing larger-scale programs by transforming high-level specifications into executable code. Like our approach, they seek to extend transformation techniques to larger-scale systems, and similarly involve the software designer in the design of data representations (usually in order to avoid limiting the expressiveness of the specification language). The developers of CIP [1], for example, advocate using algebraic specifications as the starting point for a top-down method of program development. The developers of Extended ML [19] and VDM [2] use formal verification to invent new implementations and prove them correct.

Our approach differs from these in its support for the integration of separate components. This gives the designer the flexibility to delay agreement on, as well as adapt, the interfaces of a (module) system. Goguen [10] has studied the issue of component integration for large-scale systems, and proposes a module interconnection language with a program methodology for composing software components that facilitates reuse. We speculate that it may be possible to embed our methods for integrating multiple data representations into his system.

The *views* approach of Garlan [9] has motivations similar to ours; rather than having to decide in advance on some compromise representation, separate components with appropriate representations are designed and later integrated. This allows agreement on interfaces to occur later in the design process. Unlike our approach, however, merging is restricted to a small number of fixed data types, thus yielding a greater degree of automation at the expense of expressiveness, power, and flexibility. The *programming with views* approach of the Gandalf group [11] extends Garlan's work to support the merging of arbitrary abstract data types (that are connected via "compatibility maps"), but is less automatic since it requires all operations to be rewritten by hand for a merged type. Our techniques for deriving module interfaces may provide a basis for formalizing this process of merging.

The example sketched in this paper, in which a buffer data structure for a text editor is derived from a collection of simple components, is a first step of experimentation in exploring the applicability of data transformation techniques in the management of larger-scale software systems. The example shows how module interfaces can be systematically derived in an iterative way. The formal manipulations are generally carried out within narrow syntactic contexts. There are, nonetheless, a small number of cases where more global transformations are made, but the areas of heuristic difficulty are almost entirely associated with the more narrow contexts. The examples create optimism that scaling up to apply formal program manipulation methods to larger software systems is possible, but it is also evident that this scaling up will most likely require additional techniques such as the data-type transformations illustrated in this paper.

References

[1] F.L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations — computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 1988.

[2] D. Bjørner and C.B. Jones. *The Vienna Development Method: the Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[3] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin/Cummings, Menlo Park, CA, 1987.

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

[5] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[6] R.M. Burstall and Joseph A. Goguen. Putting theories together to make specifications. In *Proceedings of Fifth International Joint Conference Artificial Intelligence*, pages 1045–1058, 1977.

[7] John Darlington. The design of efficient data representations, 1980.

[8] Martin S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens, editor, *Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Toelz, FRG*. North-Holland, November 1986.

[9] David Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie Mellon University, 1987. Available as Technical Report CMU-CS-87-147.

[10] Joseph A. Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986.

[11] A.N. Habermann, Charles Krueger, Benjamin Pierce, Barbara Staudt, and John Wenn. Programming with views. Technical Report CMU-CS-87-177, Carnegie Mellon University, Computer Science Department, January 1988.

[12] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[13] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Thirteenth Symposium on Principles of Programming Languages*, pages 86–96. ACM, January 1986.

[14] Ulrik Jørring and William L. Scherlis. Deriving and using destructive data types. In *IFIP TC2 Working Conference on Program Specification and Transformation*. North-Holland, 1986.

[15] Barbara Liskov. Abstraction mechanisms in Clu. *Communications of the ACM*, 20(8):564–576, August 1977.

[16] Barbara Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5):17–34, May 1988.

[17] Robin Milner. The Standard ML core language. *Polymorphism*, II(2), October 1985. Also Technical Report ECS-LFCS-86-2, University of Edinburgh, Edinburgh, Scotland, March 1986.

[18] Robert L. Nord. Deriving and manipulating module interfaces. Ergo Report 89–081, Carnegie Mellon University, Pittsburgh, September 1989.

[19] Donald Sannella and Andrzej Tarlecki. Toward formal development of ML programs: foundations and methodology. Technical Report ECS-LFCS-89-71, University of Edinburgh, 1989.

[20] William L. Scherlis. *Expression Procedures and Program Derivation*. PhD thesis, Stanford University, August 1980. Available as Technical Report Stan-CS-80-818.

[21] William L. Scherlis. Abstract data types, specialization and program reuse. In *International Workshop on Advanced Programming Environments*. Springer-Verlag LNCS 244, 1986.

[22] David S. Wile. Type transformations. *IEEE Transactions on Software Engineering*, SE-7(1):32–39, January 1981.

[23] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, third edition, 1985.