Specification Level Interoperability

Jack C. Wileden*
Alexander L. Wolf†
William R.Rosenblatt*
Peri L. Tarr*

COINS Technical Report 90–124
December 1990

**Software Development Laboratory*
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

†AT&T Bell Labs
600 Mountain Avenue
Murray Hill, New Jersey 07974

This paper to appear in
Communications of the ACM
March 1991

At the University of Massachusetts, this work was supported in part by National Science Foundation grant CCR-8704478 with cooperation from the Defense Advanced Research Projects Agency (ARPA order 6104).

Specification Level Interoperability

Jack C. Wileden[†]
Alexander L. Wolf[‡]
William R. Rosenblatt[†]
Peri L. Tarr[†]

[†]*Software Development Laboratory*
Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

[‡]AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, New Jersey 07974

Abstract

There is an increasing need and desire to develop systems by combining components that are written in different languages and/or that are run on different kinds of machines. Success at this depends in large part on the *interoperability* of the components—that is, the ability of the components to communicate and work together despite their differing backgrounds. While most previous approaches to interoperability have provided support at the representation level, we are pursuing an approach that provides support at the specification level. We have developed a model of such support that consists of four components: 1) a *unifying type model*, which is a notation for describing the entities to be shared by interoperating programs; 2) *language bindings*, which connect the type models of the languages to the unifying type model; 3) *underlying implementations*, which realize the types used by the different interoperating programs; and 4) *automated assistance*, which eases the task of combining components into an interoperable whole. In this paper we discuss the representation level and specification level approaches to interoperability, describe our current prototype realization of the specification level approach and our experience with its use, and outline our plans for extending both the approach and its realization.

An earlier version of this paper appeared in **Proceedings of the 12th International Conference on Software Engineering**, Nice, France, March 1990.

At the University of Massachusetts, this work was supported in part by the National Science Foundatation (CCR-87-04478) with cooperation from the Defense Advanced Research Projects Agency (ARPA order 6104).

1 Introduction

Support for interoperability is becoming increasingly important with the advent of more and larger evolving, heterogeneous computing systems. By *interoperability* we mean the ability of two or more programs to communicate or work together despite having been written in different languages or language dialects. Frequently, interoperability also implies communication between two or more different execution domains, which may range from different run-time support systems within a single processor to physically distinct processors within a distributed computing system.

The need for interoperability arises in many contexts. Generally, the desire to combine programs written in different languages springs from the availability of specific capabilities in some particular language, processor or existing program. Thus, for example, the number-crunching power of a vector processor and the availability of a particular numerical analysis routine in FORTRAN might entice a programmer to attempt interoperation of a LISP program running on a workstation and FORTRAN code running on the vector processor.

A central issue in supporting interoperability is achieving *type compatibility* so that entities, such as data objects or procedures, used in one program can be shared by another program that may be written in a different language or running on a different kind of processor. While most previous approaches to interoperability have provided support for type compatibility at the representation level, we are pursuing an approach that will support compatibility defined at the type specification level. Representation level interoperability (RLI) defines type compatibility in terms of the *structure* (representation) of objects and provides a means for overcoming differences in the ways that different programming languages or machines implement simple types. Thus, RLI hides such differences as byte orders, floating-point precisions or array accessing mechanisms. Specification level interoperability (SLI) extends RLI by defining type compatibility in terms of the *properties* (specification) of objects and hiding representation differences for abstract types as well as simple types. For instance, where RLI would hide the byte orders of array elements used to

represent a stack object, SLI would hide the fact that the stack was represented as an array. Hence, with SLI, representation of the stack as an array or as a linked list or both is made irrelevant to the interoperability of the programs sharing the stack.

Thus, SLI extends RLI in ways that offer several significant advantages. Most importantly, it facilitates integration of interoperating programs by allowing them to communicate directly in terms of higher level, more abstract types. It also increases the degree of information hiding, thereby reducing the extent to which interoperating programs depend on low-level details of each others' data representations. Moreover, it prevents the misuse of shared data objects (i.e., violations of the intended abstractions), which could easily occur under the RLI approach. SLI allows much greater flexibility in implementation approaches and hence more opportunities for optimization. Finally, SLI increases the range of languages and types that can participate in interoperation.

In this paper we present the SLI approach, give a general model of support for SLI and describe our prototype realization of the SLI approach. We begin by discussing our motivating example, namely interoperability in software environments. We then describe the representation level and specification level approaches and related work in this area. Next, we present our general model of support for SLI, a description of our initial prototype, and a report of some actual experiences with SLI and our prototype. We conclude with a discussion of future directions for work on SLI.

2 Interoperability in Environments

Our work on support for interoperability originated as part of our research on object management for next generation software development environments [9, 34]. This research is being done as part of the Arcadia project [30], a collaborative software environment research program encompassing groups at several universities and industrial organizations. The objective of Arcadia is to develop advanced software environment technology and to demonstrate this technology through prototype environments.

Three important goals of next generation software development environments, such as those envisioned by Arcadia, are extensibility, integration and broad scope. In particular, Arcadia environments are intended to be extensible in order to support experimental investigation of software process models and evaluation of novel tools in the context of a complete environment. At the same time, Arcadia environments must remain integrated, both externally, to aid users of the extended functionality, and internally, to facilitate tool cooperation, environment maintenance and further extension. Arcadia environments are also intended to be broad in scope, i.e., to support a wide variety of development activities, not merely monolingual program development and execution, and therefore to include many different kinds of tools and objects.

These goals require that Arcadia environments facilitate the addition, modification and replacement of any and all kinds of environment components, which are such things as tools, management data or process descriptions. This, in turn, leads us directly to the need for interoperability support as an important component of the Arcadia object management infrastructure.

For example, building prototype components for Arcadia has frequently led to a need for interoperation between programs written in different languages. Most often this has arisen when a tool written in Ada has needed the capabilities available in a utility written in C, such as a window manager or an object storage manager. The situation illustrated by a prototype version of the CEDL Constrained Expression Toolset [4], which implements a technique for analyzing behavioral properties of concurrent software systems, is slightly more complicated. The prototype toolset, as shown in Figure 1, includes of a *deriver*, written in Ada, that produces constrained expression representations of concurrent system behavior from system descriptions in an Ada-like design language called CEDL. The constrained expressions that it produces are then operated upon by either an *inequality generator* or a *behavior generator*, both of which are written in LISP. The inequalities generated by the former are input to an *integer programming package*, which is written in FORTRAN.

We expect that such multilingual interoperating sets of tools will be very common in the coming genera-

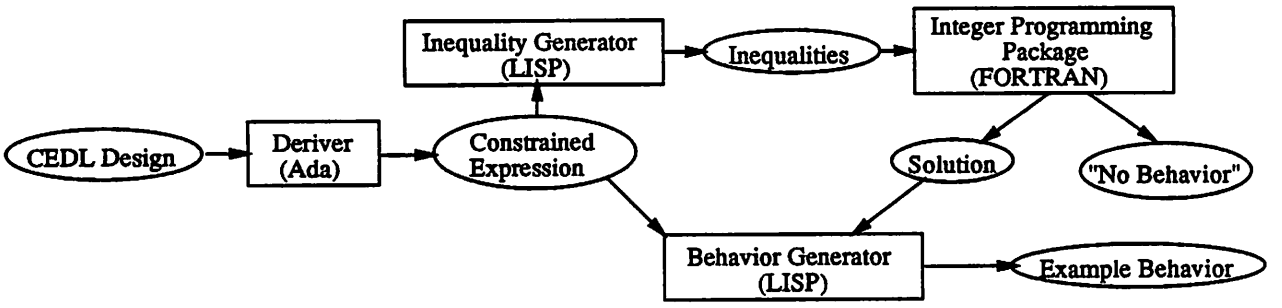

Figure 1: An Example of Multilingual Interoperability: The CEDL Constrained Expression Toolset Prototype.

tion of integrated, extensible, broad-scope environments like those envisioned by Arcadia. The need to import existing tools or utilities will be an especially significant reason for supporting this kind of interoperability.

Interoperation between programs running on different processors in a distributed computing system is also likely to be a necessity in the next generation of environments. Environments, like other complex, multi-faceted software, will need to exploit the specialized capabilities of powerful numerical processors, graphics stations, storage servers or other such hardware in a distributed system. In building prototype components for Arcadia we have already encountered several situations where interoperation across multiple machines in a distributed system was necessary. For example, the prototype Constrained Expression Toolset cannot currently be run on any single processor in our computing system because the Ada system needed for running the deriver and the LISP system needed for running the inequality generator and behavior generator do not coexist on any of our workstations. In the future we anticipate even more opportunities for interoperation across different processors to arise within Arcadia. For instance, future versions of the Constrained Expression Toolset might use specialized numerical processors for running the integer programming component.

3 Two Classes of Interoperability Issues

Any approach to supporting interoperability is based on achieving compatibility between interoperating components. There are two broad classes of concerns:

- *execution model (control) issues:* How is the execution of the interoperating programs coordinated?
- *type model (data) issues:* How are correspondences established among the ways interoperating programs manipulate a given shared entity?

Execution model issues include simple incompatibilities, such as the difference between using functions or procedures as the primary (or sole) construct for subprograms. More serious execution model issues arise in trying to interoperate sequential and concurrent programs or programs based on different concurrent communication constructs (e.g., synchronous vs. asynchronous or symmetric vs. asymmetric communication primitives). Execution model issues become most problematic if the interoperating programs are based on very different underlying execution models, such as a dataflow model and a logic programming model.[1]

Recently, a variety of interesting approaches to these execution model issues have begun to be explored. For instance, the selective-broadcast communication mechanism of Field [26] is expressly aimed at addressing execution model aspects of integration in programming environments with its simple model of program interaction based on message-passing semantics. Similarly, the software bus (or "toolbus") mechanism in Polylith [25] offers an encapsulation of interprocess communication protocols that is intended to simplify interconnection of components in multilingual software systems.

While execution model issues pose some potentially challenging interoperability problems, our work to date has focused on type model issues. We have implicitly relied upon the adequacy of the ubiquitous procedure call as a fundamental building block from which most (sequential or concurrent) control constructs

[1] It could be argued that both of these examples, as well as such mechanisms as "active data" or "triggers", represent intermingling of execution model and data model issues. It is possible that some perspective that separated those components would make such mechanisms fit more smoothly into the framework presented here.

can be synthesized. This seems reasonable, since most existing approaches to support for interoperability have been based on the use of the *remote procedure call* (RPC) [7] for coordinating the execution of the interoperating programs. Such mechanisms are in common use today (e.g., [5, 22, 29, 33]). Of course, RPC mechanisms involve some type model issues as well (see below).

4 Existing Approaches to Type Model Issues

Interoperability depends fundamentally on determining and achieving type compatibility. That is, when two interoperating components are sharing, or communicating via, some data object, they must have consistent views of whatever properties they mutually rely upon that are associated with objects of that type.[2] Note, however, that the need for compatible type definitions does not necessarily imply a requirement for identical type definitions. In the Constrained Expression Toolset example, for instance, the deriver, inequality generator and behavior generator must all have compatible views of the type of the constrained expression objects that they share. Their definitions (views) of that type needn't be identical, however, but simply sufficiently similar to allow them to communicate correctly and unambiguously.

There are a variety of existing approaches to the type model aspects of interoperability. Most of these approaches have been based on establishing compatibility (and usually identity) of data types at the representation level. As indicated earlier, we believe that there are significant advantages to addressing these issues at the type specification level rather than only at the type representation level. In the remainder of this section, we survey existing approaches to type model aspects of interoperability.

[2]The object-oriented type model notion of *conformance* between a subtype and its supertype(s) is a special case of the notion of type compatibility.

4.1 Single Type Model

An obvious approach to type compatibility in interoperating programs is to impose a single type model on all the languages in which interoperating programs are to be written. Since the type definitions of all entities to be shared by all interoperating programs are then, necessarily, directly comparable, establishing the necessary type compatibilities (e.g., by insisting on identical type definitions) is straightforward. Of course, such an approach is tantamount to imposing a single programming language on the implementors of all potentially interoperating programs. To apply it to our Constrained Expression Toolset example, for instance, we might recode all of the toolset's components in a single language. Once the whole toolset was written in Ada (or LISP, or FORTRAN, or whatever), it would be trivial to establish type compatibility of the shared data objects.

This approach has generally been proposed in the context of multi-machine interoperability. For example, Herlihy and Liskov [14] have taken this approach using the CLU language, though they suggest that it can be used with any language that supports abstract data types. Similarly, Emerald [8] is a language that supports multiple representations of objects across different machines, but again the assumption is that all interoperating programs are written in the same language, namely Emerald. While this approach clearly solves the type compatibility problem, it obviously doesn't address our goal of interoperability in multi-lingual systems, such as next generation software development environments.

4.2 Single Universal Representation

Perhaps the most widespread, traditional approach to type model aspects of interoperability is based on explicitly translating shared objects to and from a single universal representation, such as characters or bytes. The earliest form of this approach involved interoperation through ASCII representations of data, where the data were communicated between the "interoperating" programs via files. This required the programs

themselves to translate the data either into or out of the ASCII representation. Of course, languages often provided some automated support for this processing (e.g., the FORTRAN format statement).

The UNIX[3] operating system supports interoperability via *pipes* (untyped byte streams) through which two interoperating programs can communicate. The byte stream can encode any type of data. So, as long as the interoperating programs agree on how to interpret the bytes, they can share data of any type. Again, this requires the programs at either end of the pipe to translate a shared object from its actual type into the byte stream representation and back again. In our Constrained Expression Toolset example, for instance, the deriver might encode a constrained expression as a sequence of characters such as:

(* (V E1 E2 E3))

which would then be decoded by the inequality generator or the behavior generator.

Under this approach, data type compatibility is determined by the compatibility of the two translations (into and out of the universal representation).

4.3 Standardized Basic Types

A more recent and more sophisticated version of the single universal representation approach is the use of a standardized set of definitions for a set of basic types, such as integer, float, and string. Approaches of this kind were associated with early RPC mechanisms, which sometimes leads to a confusion between RPC (one approach to address execution model concerns) and interoperability. Because the main problem early RPC mechanisms tried to address was different hardware representations of data on different machines (e.g., byte orderings, floating-point formats, character encodings, etc.) it is not surprising that they focused on basic, directly hardware supported, types. Later versions of this approach have added support for standard

[3]UNIX is a registered trademark of UNIX System Laboratories, Inc.

aggregate type constructors, such as arrays or records.

When this approach was employed in conjunction with the earliest RPC-based systems, it only allowed data types within a single, fixed language domain to be passed from process to process. Thus, these RPC mechanisms not only abstracted away details of the communication protocol (e.g., below the ISO transport layer), but they also abstracted away representational discrepancies among machines and within the fixed language domain (e.g., byte-orderings, floating-point precisions). Support for this independence of representation was not totally automated in some earlier efforts (such as those cited above); the code that translates between physical representations needed to be written by hand.

More recent RPC mechanisms, such as NCS [3] and HRPC [6], automate the generation of code that maps the representations of data types within a program from one machine to another via an intermediate interface description language such as HP/Apollo's NIDL [3] or Xerox's Courier [38]. Although these two systems can only generate interface code in a single language (C in both cases), their approach seems to lend itself well to supporting additional languages without too much extra effort.

Several systems, such as MLP [13], Q [21], Horus [12], and Matchmaker [15], have taken the additional step of providing RPC-based support for mixed-language programming. For example, MLP defines a Universal Type System (UTS) for describing objects that are passed among programs in various language domains. UTS builds up types from primitives such as "integer" and "float" via constructors such as "array" and "record" in a manner similar to what would be done in a language like Pascal. It communicates data among language domains by providing a small set of standard routines that must be implemented for each language. Programmers use these routines within their code to translate from their language domains to UTS and vice versa. MLP was not designed with a particular set of languages in mind and, therefore, it appears to be applicable to a fairly wide variety of languages. The same is true of Horus and Matchmaker.

Q, on the other hand, is designed specifically to support interoperation between C and Ada programs across a heterogeneous network. It is an explicit extension of Sun's XDR/RPC, an existing RPC mechanism

that only supports C data types. Like MLP, Q also uses a "constructive" approach to building up the values that are transmitted between programs. Another, similar attempt at multilingual interoperability is the Mercury project [18], which is designed to support interoperability among the C, LISP and Argus [19] language domains.

Under the Standardized Basic Types approach, whether or not it is extended with aggregate type constructors, data type compatibility is determined via the equivalence of the structures defined for a type in all of the programs that share that type. In our Constrained Expression Toolset example, for instance, the deriver might define the constrained expression type to be an Ada record structure, the behavior generator and inequality generator might define it with a Common LISP **defstruct**, and the compatibility of these definitions would depend upon the field-by-field equivalence of the record and **defstruct** definitions.

4.4 The Common Theme: Representation Level Interoperability (RLI)

The Single Universal Representation and Standardized Basic Types approaches are paradigmatic cases of *representation level interoperability*. In an RLI approach:

- type modeling is based on the structure (representation) of objects, and
- type compatibility is based on comparison (or explicit translation) of structures (representation).

Representation level support for interoperability is both useful and necessary. In our view, however, it has several shortcomings. Chief among these is the fact that RLI is only applicable to low-level simple types (e.g., integers) or compound simple types (e.g., arrays of integers). In particular, RLI does not support abstract types, such as "stack" or "abstract syntax tree". This not only makes RLI awkward to use in conjunction with modern languages having rich and extensible typing mechanisms (Ada, C++, CLOS, etc.), but also leads to low-level dependencies on type representations between interoperating programs. For example, if two programs employ an RLI approach to sharing a stack data object, then both programs would be forced

to use the same representation of the stack, such as an array with an integer index pointing to the "top" item, or as a linked list of records of a particular form.

Furthermore, RLI limits the flexibility and extensibility available in interoperating systems. Its reliance on isomorphism of low-level structures inhibits interoperation through similar but not identical types of entities and eliminates any possibility of using different underlying representations for different instances of the same type, thus foreclosing opportunities for optimization of representations.

4.5 Single Standardized Submodel

A final approach to type compatibility in interoperating programs is typified by a database style of interaction among components. In the Single Standardized Submodel approach, all interoperating programs must use a single type model, distinct from those found in the languages in which the programs are written, to describe any shared objects. But unlike in the Single Type Model approach, unshared objects can be described using the type model(s) of the host language(s) of the interoperating programs. In traditional forms of this approach, the shared submodel provides some basic types, such as numbers and strings, and a very limited set of aggregation constructors, such as tuple or relation. Interoperating programs then carry out all manipulations of shared objects through a foreign language utilizing type definitions formulated using the shared submodel, such as an embedded query language (e.g.,[28]). IDL [1] represents another form of this approach, in which the constructors are attributed graph, set, sequence, and node. More recently, object-oriented databases (e.g., [2], [11], [17], [20]) have begun to offer richer type submodels.

The Single Standardized Submodel approach to type compatibility has been at the core of several proposals explicitly aimed at supporting interoperability in software development environments. In particular, CAIS [23], PCTE [31], and the Atherton backplane [24] have all been based on the Single Standardized Submodel approach.

For abstract types, such as the constrained expression type in our Constrained Expression Toolset example, the Single Standardized Submodel approach still provides representation level support for interoperability. Now compatibility is based on a single type submodel and hence a single structure (representation), but abstract types will still have to be encoded using the basic types and aggregation constructors supplied by that submodel. Thus the Single Standardized Submodel approach has some of the characteristics of the Single Type Model approach and some characteristics of the Standardized Basic Types approach.

5 Specification Level Interoperability (SLI)

The SLI approach is motivated by our belief that developers of potentially interoperating programs should have the maximum possible flexibility, convenience and range of expressive power available when defining types for the objects that their programs will manipulate. In particular, they should be free to create those type definitions in terms of the type models found in the languages in which their programs are written. They should have full power to develop and use appropriate abstract types, and maximum freedom to ignore the representation (implementation) details associated with those type definitions. Finally, the distinction between shared and unshared objects should have minimal impact on the program and its developer. Specifically, neither the fact that an object is expected to be shared (or not to be shared) among interoperating programs, nor a later change in that status, should affect the interface to that object as seen by the rest of the program that manipulates the object. An interoperability approach lacking these features will severely impede integration and extensibility. Unfortunately, RLI approaches are deficient in all of these areas.

Specification level interoperability overcomes the shortcomings associated with representation level interoperability. Rather than focusing on the mapping between different representations of a type, SLI focuses on support for common definitions of a type's properties. The SLI approach thereby attains the benefits of abstraction and information hiding for interoperating programs, encouraging the use of entity descriptions

(i.e., type definitions) that promote the overall organization of a software system. By raising the level of cooperation from isomorphism of representation to equivalence of overlapping properties of shared types, SLI eliminates low-level dependencies among interoperating programs, enables interoperation through similar but not identical entity types, and permits differing representations for different instances of a type, thus allowing for optimized representations. Of course, SLI depends upon RLI mechanisms, essentially subsuming RLI in those cases involving simple types.

In pursuing the goal of specification level interoperability, we began by developing a general model of the support required to realize SLI. Guided by the model, we then assembled a prototype realization of SLI as a demonstration of its feasibility and usefulness. We now proceed to describe both of these aspects of our work on specification level interoperability.

6 A Model of Support for SLI

Our model distinguishes four components necessary to fully support specification level interoperability:

1. *A Unifying Type Model (UTM):* A UTM is a notation for describing the types of entities that are to be shared by interoperating programs. UTM type definitions supplement but do not replace the type definitions for the shared entities that are expressed in the language(s) in which the interoperating programs are written. A UTM must be a *unifying* model, in the sense that it is sufficient for describing those properties of an entity's type that are relevant from the perspective of any of the interoperating programs that share instances of that type. Hence, a UTM should be capable of expressing high-level, abstract descriptions of the properties of a broad range of types, but need not adhere too closely to the syntax or type definition style of any particular programming language.
2. *Language Bindings:* Given a UTM and a particular programming language, there must be a way to relate the relevant parts of a type definition given in the language to a definition as given in the UTM. Each such mapping between a UTM and a particular language is referred to as a *language binding*. Note that not all aspects of a UTM must be mappable to a given language, but only those that are, or could be, relevant to programs in that language. Hence, a set of different bindings could be defined for a given language, each providing mappings for only those UTM aspects relevant to a particular interoperating program written in that language.
3. *Underlying Implementations:* The combination of a UTM type definition and a language binding induces an *interface* through which an interoperating program written in that language can manipulate

instances of the entity type. Underneath the interface will be one or more *representations* for data objects and *code* to implement procedures, such as the procedures (i.e., operations) that the interface provides for manipulating the data objects. A major benefit of the SLI approach is that all such details of implementation are hidden from the interoperating programs by the interface. This permits experimentation with alternative implementations, "rapid prototyping" development styles in which "quick and dirty" implementations can later be improved without affecting the interoperating program, or even heterogeneous representations for the same type so that different instances can be optimized for different kinds of manipulations (e.g., navigational access vs. associative access to different collections of data of some particular kind).

4. *Automated Assistance:* Although SLI can be beneficially employed using entirely manual methods, its value is greatly increased through automated support. In particular, someone creating a UTM definition would be greatly aided by a *library* of pre-existing UTM type definitions, language bindings and underlying implementations, plus a *browser* for exploring that library. An *automated generation* tool would also be valuable. Such a tool would, for example, take a UTM type definition, plus specifications for the desired language binding and underlying implementation (possibly indicated interactively through a selection capability in the browser), and generate the corresponding interface.

Following is a scenario that illustrates how we envision a full-scale realization of this model being used. The next section describes our initial prototype, which provides a subset of the capabilities listed above.

Suppose that a newly developed program is to be integrated with an existing set of programs. The implementor of the new program might begin by determining which objects used by which programs in the existing set would need to be shared with the new program. In the case of objects that are already shared among the existing programs or whose sharing had been anticipated by their developers, UTM definitions for their types might be found by browsing the type definition library. In other cases, new type definitions might be created using the UTM notation, possibly by finding and modifying or extending existing definitions from the library. In a similar fashion, the implementor would find or create appropriate language bindings for the language(s) in which the program is to be implemented, and find or create suitable implementations.

Using the automated generation tool, the implementor would then produce the necessary interface in the implementation language selected for the program, plus any representation and code needed to effect the implementation of object instances. The representation and code could be as simple as RPCs to operations that were part of the existing implementations of the object types in the existing set of programs. In

such a case, the automated generation tool could produce them completely given only the specification stating that this was the desired implementation. Much more complex situations could also be supported. For example, a completely new implementation could be defined with corresponding modifications to the representation or code attached to the pre-existing programs' interfaces to objects of this type. Alternatively, two parallel implementations could be utilized, one associated with each of the programs sharing objects of this type, with consistency maintenance mechanisms and bi-directional translation linkages joining the two implementations. In such situations, the automatic generation tool's role would be more limited, but the role of the library and browser in aiding future implementors wishing to interoperate with objects of this type would be correspondingly more important.

7 An Initial Prototype Realization of SLI

To demonstrate the feasibility of SLI and to support experimentation with the model of SLI presented in Section 6, we have developed an initial prototype realization of the model that provides a subset of its capabilities. This prototype realization consists of a first approximation to a unifying type model, called UTM-0, bindings for the LISP and Ada programming languages, one implementation strategy, and the UTM-0 automated generator shown in Figure 2. The automated generator accepts type definitions in the UTM-0 notation, plus binding and implementation information, and produces a *standard interface specification* (SIS) and a corresponding implementation for each of the entities described in the UTM-0 input. By "standard" we mean that, for a given target programming language, the UTM-0 automated generator will always produce the same interface specification from the same UTM-0 input. In fact, due to the bindings defined in this prototype, the LISP and Ada SISs that are generated from a given UTM-0 description are essentially identical, differing only in the syntax used to express them. The implementations attached to those SISs by the automated generator may vary widely, depending upon the implementation information that was provided with the UTM-0 definitions. The implementations, of course, are completely invisible to

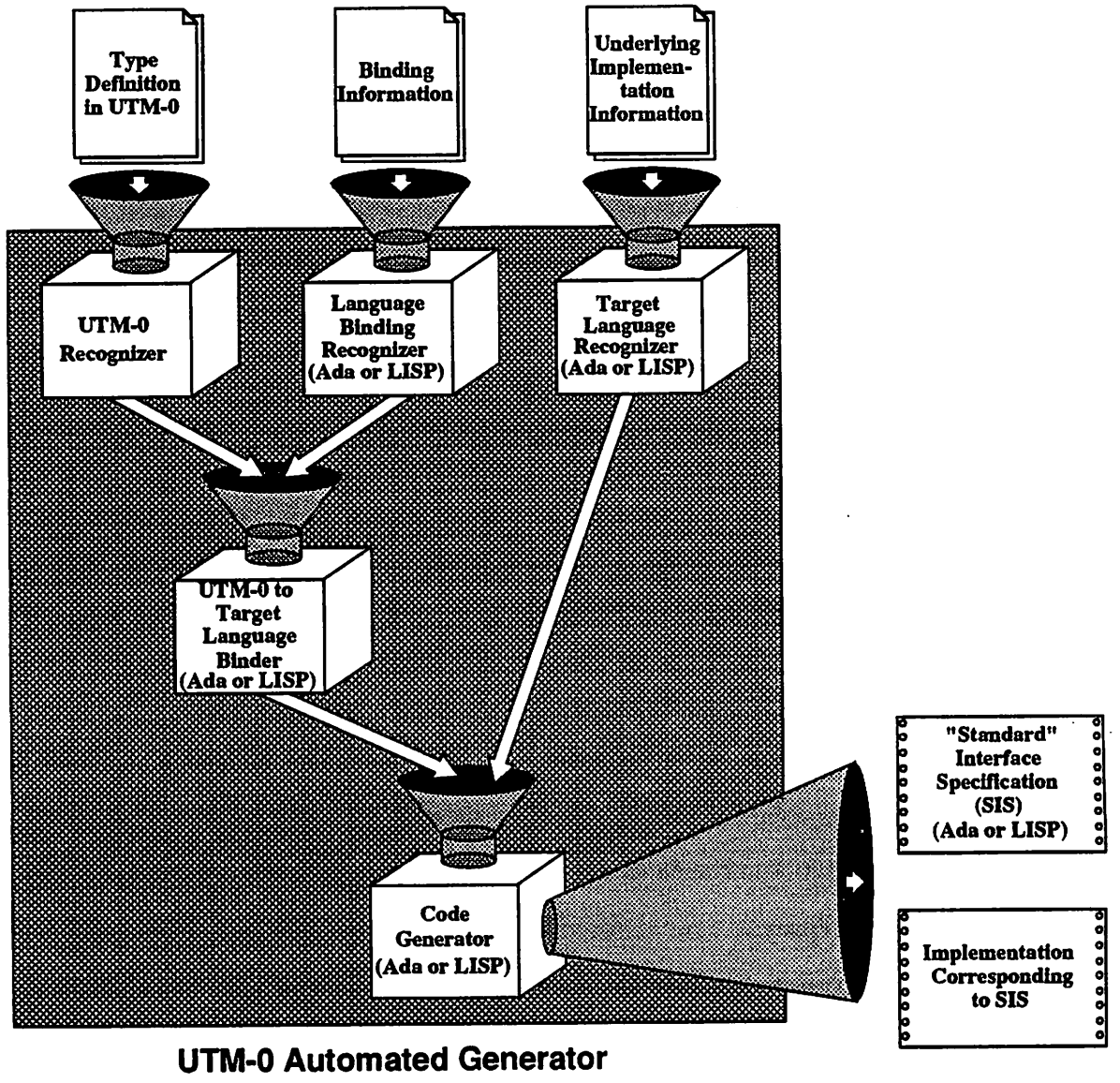

Figure 2: The UTM-0 Automated Generator.

the interoperating programs, which need only be concerned with the SISs.

In this section, we describe each of the components of our initial prototype realization of SLI. We illustrate the use of these components through an example in which we applied the UTM-0 automated generator to

achieve interoperability between some of the components of the Constrained Expression Toolset described in Section 2.

7.1 Definition of UTM-0

UTM-0 consists of a set of type definition primitives, a set of "special" types, and some semantics for manipulation of instances of types. We present a brief overview of UTM-0 here (a more detailed description can be found in [36]) and an example of its use in Section 7.2.

The type definition primitives of UTM-0 are based on the approach used in the OROS type model [27] and are similar to those found in other recently proposed type models (e.g., [2, 10, 32, 39]). Like OROS, UTM-0 distinguishes three basic classes of types: object types, relationship types and operation types. Intuitively, object types are used to describe things whose state is their most interesting property, operation types are used to describe things that manipulate or transform other things, and relationship types are used to describe things that represent connections among other things. UTM-0 uses the word "entity" to encompass all things; thus object types, relationship types and operation types are all also entity types.

Using UTM-0, types are defined in terms of a set of properties and in terms of their relationships to other types (*intertype relationships*). The properties of a type are the operations that can be applied to its instances, relationships in which its instances can participate, and possibly a signature.[4] The intertype relationships include inheritance of properties, explicit differentiation of properties relative to those of other types, and subtyping. Types defined using UTM-0 have names and can be parameterized.

UTM-0 supports multiple inheritance, with the **parents** field of a type definition listing the inherited types. The UTM-0 rules for inheritance are fairly simple. They are as follows: if type T lists type P as a

[4]The signature of a relationship type describes the number, types and modes of the entities connected by an instance of the relationship, while the signature of an operation type describes the number, types and modes of parameters to the operation. Object types do not have signatures in UTM-0.

parent, T inherits P's parents (recursively) and all of P's associated operations and relationships. If P has a signature, it is prepended to T's signature (if any). If there is a naming conflict among multiple parents, the parent listed first prevails. The UTM-0 model also supports the concept of subtyping, whereby if type S is a subtype of T (S conforms to T), an instance of type S can be used wherever an instance of T is called for. The UTM-0 subtyping rules, which are based on those of OROS, are detailed in [36].

The remainder of the UTM-0 definition consists of definitions of some *special types*, criteria for type equivalence, and semantics for instance manipulation.

The most important special types in UTM-0 are the four *primitive types* in terms of which all other types are defined, namely entity, object, relationship and operation. Their definitions, in turn, involve some further operation and relationship types. Since they are used in defining the primitive types, these types are also considered "special" and are referred to as *primitive operation* and *primitive relationship* types. The final category of special types is the *simple object* types, which can include such commonly used object types as integer, character and real. Just as the primitive operations are subtypes of operation, the simple object types are subtypes of object. Simple object types differ from all others in that they have more limited semantics.

UTM-0 defines a fairly simple semantics for manipulation of instances of types. A variable of a simple object type holds a *value* (e.g., the integer 17), while variables of other entity types hold *pointers* to the entities themselves. Semantics for assignment and entity equivalence follow directly from this dichotomy. Assignment for simple objects is done by copying their values, while assignment for other entities is done by copying pointers. Similarly, equivalence among simple objects implies equal values, while among other entities it implies equal pointers.

Finally, the UTM-0 definition includes a library of useful type definitions. This "standard library" includes simple object types such as integer, character and real. It also includes several parametric types that define commonly-used aggregates, such as array[T], relation[T] and sequence[T]. Hence the UTM-0 standard

library provides a superset of the type definition capabilities available in an RLI type description language such as NIDL or Courier.

7.2 UTM-0 Example

The Constrained Expression Toolset is a collection of tools that must interoperate to perform their individual tasks. As discussed in Section 2, the deriver component, which is written in Ada, accepts a CEDL description of a concurrent system and produces a constrained expression representation of the system in the form of a tree (a *constrained expression abstract syntax tree*, hereafter abbreviated CEAST). The behavior generator and inequality generator components are written in Common LISP, and each of these tools uses the information stored in the CEAST. Because the CEAST created by the deriver is an Ada object, however, it had been impossible for the LISP tools to manipulate the tree directly.

Originally, as an interim solution, another Ada tool was written to translate the CEAST into an ASCII representation of a LISP S-expression encoding of the tree, which was written to a file. The LISP tools then read the file and manipulated the S-expression. This translation was a minor variation on the Single Universal Representation approach to RLI. As an experiment in applying the SLI approach, we decided to use our initial prototype realization of SLI to generate a replacement for the interim solution. The first step in this process was to create the appropriate UTM-0 description for the CEAST. Part of that UTM-0 description appears in Figure 3.

The description includes definitions of two object types: Node, which describes properties of graph nodes in general, and CEAST_Node, which describes the properties of the specific kinds of nodes that appear in CEAST graphs. Note that CEAST_Node inherits from Node, so that the operations that apply to Node objects (creation, deletion, and determining what kind a node is) plus the additional operations that apply to nodes of type CEAST_Node (for manipulating the specific kinds of attributes that this kind of node contains) all apply to CEAST_Node objects. Also included in the UTM-0 description of CEAST nodes are definitions

```
object type Node [ ] is
   operations:
        Kind   : KindOperation;
        Create : CreateNodeOperation;
        Delete : DeleteNodeOperation;
end

operation type CreateNodeOperation is
   signature:
        ( TheKind : in String;
          TheNode: out Node )
end

...

object type CEAST_Node is
   parents:
        Node
   operations:
        GetSymbolAttribute : GetSymbolAttributeOperation;
        PutSymbolAttribute : PutSymbolAttributeOperation;
        GetSequenceOfCETermsAttribute : GetSequenceOfCETermsAttributeOperation;
        PutSequenceOfCETermsAttribute : PutSequenceOfCETermsAttributeOperation;
end
operation type GetSymbolAttributeOperation is
   signature:
        ( TheNode          : in Node;
          TheAttributeName : in String;
          TheValue         : out Symbol )
end
operation type PutSymbolAttributeOperation is
   signature:
        ( TheNode          : out Node;
          TheAttributeName : in String;
          TheValue         : in Symbol )
end
...
```

Figure 3: Part of the UTM-0 Description of CEAST.

of the various operation types that are part of these two object type definitions.

7.3 The Language Bindings

As mentioned in Section 6, there must be at least one language binding for each programming language in which interoperating programs are written. Each language binding maps UTM definitions to syntactic constructs within a particular language. This essentially comes down to determining which language constructs correspond most closely with object, operation, and relationship type definitions. We have defined two language bindings for our prototype so far: one for Ada and one for Common LISP.

Because Ada is so supportive of abstract type definitions, it was relatively easy to produce an Ada language binding. We define all object and relationship types to be *private* types. Operation types are represented as either procedure or function declarations, depending on how the user specifies the code that implements them. Since object and relationship types are declared to be private, their representations are completely hidden from the user, which makes providing low-level representational optimizations transparent. Figure 4 illustrates how the Ada binding applies to the UTM-0 description of the CEAST example.

Our language binding for Common LISP is similar to the Ada binding. We define object and relationship types with the predefined function deftype, and use defun to implement all operation types. Figure 5 illustrates how the LISP binding applies to the UTM-0 description of the CEAST example. Because abstract data typing is not a part of the Common LISP language definition, however, it is not possible to achieve the same degree of enforced encapsulation as in Ada. With disciplined use of a LISP SIS, it is nevertheless possible to obtain the same degree of representational independence. We believe that CLOS [16] provides better support for the definition of abstract types, and it is likely that we will implement a CLOS binding in the near future.

Of course, both the Ada and the LISP bindings include bindings for simple object types as well. Our initial prototype realization uses the obvious bindings for the integer and character simple object types, which are all that are needed for our first set of experimental applications. Future prototypes will include a more

```
------------------------------------------------------------
--
-- This interface was generated by the UTM-0
-- Automated Generator Version 1.0.
--
-- Input file: /u/oread/tarr/UTM/ceast.utm
-- Date: JULY 5, 1989
-- Time: 03:48:42
--
------------------------------------------------------------

package CEAST_Definitions is

  type CEAST_Node is private;
  type Symbol is private;
  type NodeKindName is new String;
  type AttributeName is new String;
  ...

  function Create ( TheKind : NodeKindName ) return CEAST_Node;
  procedure Delete ( TheNode : in out CEAST_Node );
  function Kind ( TheNode : CEAST_Node ) return NodeKindName;
  ...

  function GetSymbolAttribute ( TheNode : CEAST_Node;
                                TheAttributeName : AttributeName )
                                return Symbol;

  procedure PutSymbolAttribute ( TheNode : in CEAST_Node;
                                 TheAttributeName : in AttributeName;
                                 TheValue : in Symbol );

  ...

private
  ...
end CEAST_Definitions;
```

Figure 4: Part of the Ada Standard Interface Specification (SIS) for CEAST.

thorough treatment of bindings for simple object types, integrating more of the features found in existing *RLI type* definition mechanisms.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;
;;  This interface was generated by the UTM-0
;;  Automated Generator Version 1.0.
;;
;;  Input file:  /u/oread/tarr/UTM/ceast.utm
;;  Date:  JULY 5, 1989
;;  Time:  03:51:33
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(deftype CEAST_Node 'FOREIGN-REFERENCE)

;; Representation-level interface declarations:

...

;; Entity Declarations:

(defun Create ( TheKind )
  (call-out Ada-Create TheKind))
(defun Delete ( TheNode )
  (call-out Ada-Delete TheNode))
(defun Kind ( TheNode )
  (call-out Ada-Kind TheNode))
...
(defun GetSymbolAttribute ( TheNode TheAttributeName )
  (call-out Ada-GetSymbolAttribute TheNode TheAttributeName))
(defun PutSymbolAttribute ( TheNode TheAttributeName TheValue )
  (call-out Ada-PutSymbolAttribute TheNode TheAttributeName TheValue)
...
```

Figure 5: Part of the LISP Standard Interface Specification (SIS) for CEAST.

7.4 Generating the SISs and Corresponding Implementations

Using the straightforward mapping rules implied by each language binding, it is not difficult to generate a standard interface specification to the entities specified in a UTM-0 description. However, it is considerably more difficult to generate the actual implementations of those entities (i.e., the underlying representations of object and relationship types, and the code that implements operation types). In our initial prototype realization, entities may be implemented in one of two ways: with user-specified source code, or by interfacing

to an existing SIS for the entity that has already been processed by the UTM-0 automated generator.

In the former situation, producing an implementation corresponding to a given SIS is a relatively simple process. The UTM-0 automated generator contains an additional *language recognizer* component for each of the relevant implementation languages, which is used to parse user-supplied implementations and turn them into a standard internal representation. The parsed implementations are then used, along with the output of the language binding component, by the *language code generator* to produce the SIS and its corresponding implementation.

Figure 6a illustrates this case. To produce the Ada SIS and corresponding implementation for the CEAST type, we provided the UTM-0 automated generator with the UTM-0 description of the CEAST type and an Ada package body that implemented the types and operations specified in that description. The package body itself had been previously generated from a declarative description of the CEAST graph type using our PGRAPHITE tool [35]. The automated generator then produced the appropriate Ada SIS (Figure 4) and an implementation of that SIS using the package body that we had provided.

Generating a corresponding implementation for a given SIS when entities that it defines are to be implemented by existing SISs is handled somewhat differently. When the existing SIS is written in the same language as the one being generated, it is not difficult to "import" the existing definitions and make the appropriate operation calls. The real advantage of the automated generator, however, becomes apparent when the existing SIS is written in a different language, since this is when serious type compatibility issues arise.

Figure 6b illustrates this case. To produce the LISP SIS and corresponding implementation for the CEAST type, we provided the UTM-0 automated generator with the UTM-0 description of the CEAST type and a specification of the correspondence between various parts of the UTM-0 description and parts of the Ada SIS. Although we created the specification of the correspondence manually, a more powerful automated generator with a library capability certainly could have automated the creation of this information. The

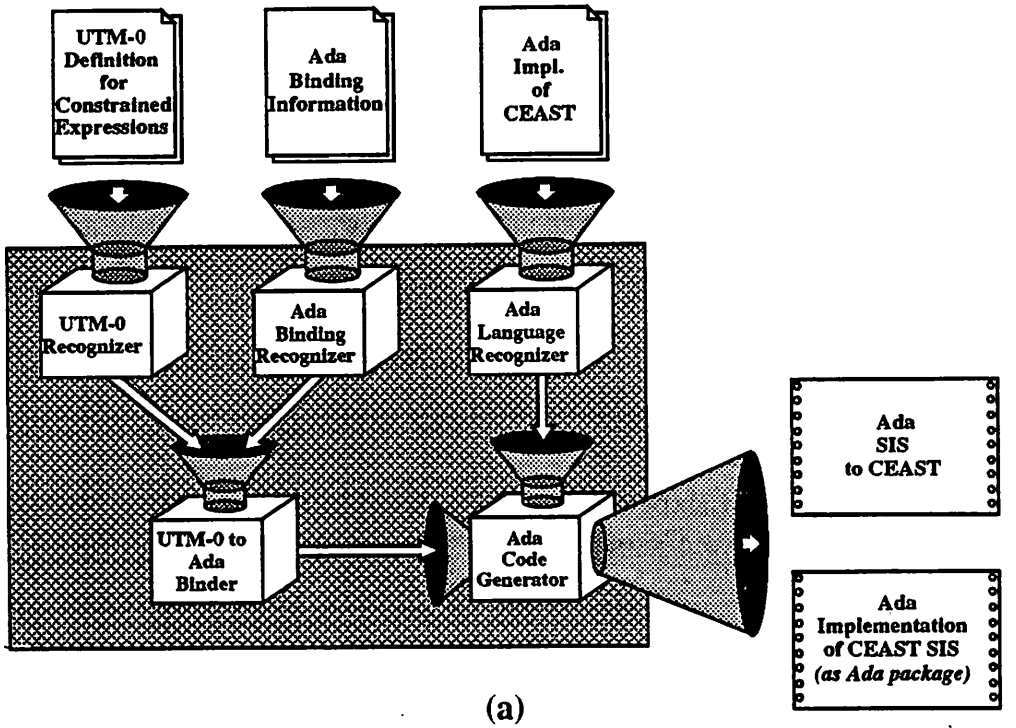

Figure 6: Using the UTM-0 Automated Generator to Produce Ada and LISP SISs and Corresponding Implementations to CEAST.

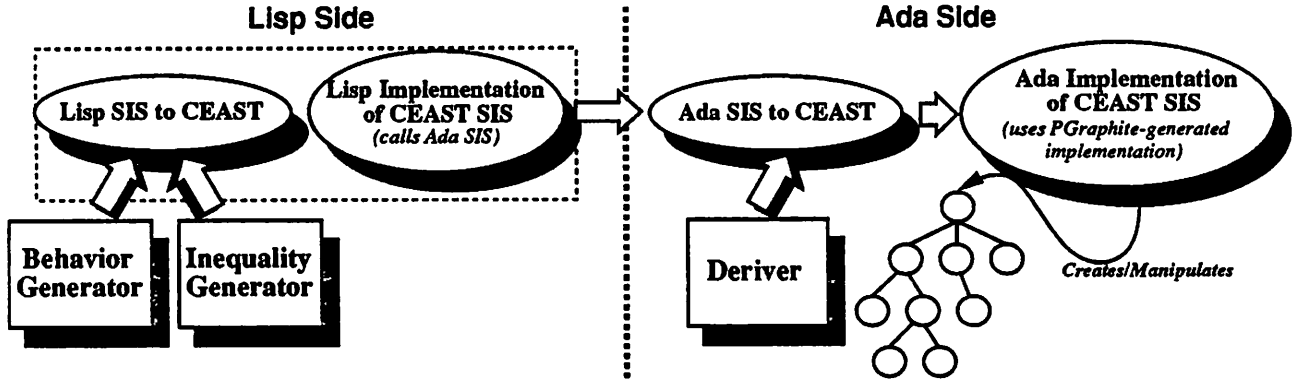

Figure 7: Constrained Expression Toolset Configuration Incorporating Interoperability Components Produced by the UTM-0 Automated Generator.

UTM-0 automated generator then produced the appropriate LISP SIS (Figure 5) and an implementation of that SIS consisting of calls to the operations provided in the Ada SIS (using the **call-out** function supported in several implementations of Common LISP).

To complete our SLI experiment, we modified the existing behavior and inequality generators to manipulate the CEAST through the LISP standard interface, instead of through S-expressions.[5] The resulting system configuration is shown in Figure 7. It is important to note that the behavior and inequality generators never "know" that they are manipulating Ada objects, nor that they are calling Ada operations. The underlying implementations are hidden from the tools, having been completely abstracted away by the SIS. This allows us the freedom to experiment with a wide variety of different implementation schemes.

At present, we employ only the implementation strategy illustrated by this example, wherein object and relationship types that do not exist locally are implemented as "foreign references" to the types de-

[5]This step was necessary because the two LISP tools had not originally been written in a style employing data abstraction. SLI is most helpful when the interoperating programs do make use of data abstraction, but different choices of language bindings and implementation strategies in our prototype could have obviated the need for these modifications to the behavior and inequality generators.

fined in the SIS where they are locally defined, and operation types are implemented as foreign operation calls. We recognize that this strategy forces an inter-language call to be made upon each and every object/relationship/operation type access, and this would probably be too expensive a solution for many applications. For example, if a LISP tool manipulates an Ada binary search tree, our implementation strategy would force an Ada operation call every time the LISP tool wanted to either examine or set values of attributes of nodes in the tree. In some cases it might be more efficient, for example, to copy the information from the Ada object to a local LISP object, which the LISP tool would then access. We have begun exploring ways to let the user select alternative, optimized implementations, but in any event, the alternative selected would be hidden from the tool.

8 Conclusion

Specification level interoperability provides a high-level, representation-independent approach to combining software components that are written in different languages or that are run on different machines. Unlike most other approaches to supporting interoperability, which focus on implementation concerns, SLI is aimed at maximizing the flexibility, convenience and expressive power available to developers of interoperating programs. At the very least, SLI can serve as a basis for the disciplined and orderly marshaling of interoperable components. If fully realized and properly used, SLI can be a type-safe, extensible mechanism offering some automated assistance in solving an important problem.

Our experiences with an initial prototype are extremely encouraging. SLI, even in the limited form delivered by the prototype, provided us with useful support in constructing a large system made up of diverse components. This suggests that the SLI approach can yield an important enabling technology for integrated, extensible, broad-scope environments in particular and large, evolving, heterogeneous software systems in general.

Our experience with using SLI and the prototype have also suggested a number of important directions for

future work. Chief among these is the need for improved UTMs and corresponding capabilities for establishing semantic equivalence between a UTM definition and the type definitions that supposedly correspond to it in each interoperating program. Where UTM-0 relies on name and signature conformance of operations and relationships in establishing such equivalence, a UTM based on richer semantic constructs (e.g., those used in Larch [37]) would provide better assurance of type compatibility. Such a UTM would also admit better automated assistance, both in compatibility checking and in automated generation, for users of SLI.

Additional work is also needed in the area of implementation strategies. We felt that the single, straightforward strategy used in our prototype was a good one for experimentation, since it simplified the problem of determining type compatibility. However, a large number of possible alternative implementations exist, including ones that would make use of RPC/XDR, NCS, HRPC, Q, IDL, Field or Polylith.

Finally, further work is also needed on language bindings and automated assistance. We are currently working on bindings for C, C++, and Prolog. C++ in particular should exercise the UTM notions of inheritance and subtyping. We are also looking into the development of a library of UTM definitions, language bindings, and underlying implementations, together with a browser for that library. Having a library and browser should make it easier to rapidly pull together the pieces needed to effect the interoperability of a given collection of components.

Acknowledgements

We appreciate the contributions made by Michel Bosco, Lori Clarke, Dennis Heimbigner, Philip Johnson, Eliot Moss, Leon Osterweil, and Stan Sutton to the work described here. We also appreciate the comments and suggestions provided by our other colleagues in the Arcadia consortium. Susan Avery, George Avrunin, John Burnett, Ugo Buy and Laura Dillon all contributed to the design and implementation of the Constrained Expression Toolset.

At the University of Massachusetts, this work was supported in part by the National Science Foundation (CCR-87-04478) with cooperation from the Defense Advanced Research Projects Agency (ARPA order 6104).

REFERENCES

[1] Special Issue on the Interface Description Language IDL. *ACM SIGPLAN Notices*, 22(11), November 1987.

[2] T. Andrews and C. Harris. Combining Language and Database Advances in an Object-Oriented Development Environment. In *OOPSLA Conference Proceedings*, pages 430–440, October 1987. Published as *ACM SIGPLAN Notices*, vol. 22, no. 12, December 1987.

[3] Apollo Computer Inc., Chelmsford, MA. *Network Computing System: A Technical Overview*, 1989.

[4] G.S. Avrunin, L.K. Dillon, and J.C. Wileden. Experiments in Automated Analysis of Concurrent Software Systems. In *Proc. ACM Software Testing, Analysis and Verification Symposium*, pages 124–130, December 1989.

[5] E. Balkovich, S. Lerman, and R.P. Parmelee. Computing in Higher Education: The Athena Experience. *Communications of the ACM*, 28(11):1214–1224, November 1985.

[6] B.N. Bershad, D.T. Ching, E.D. Lazowska, J. Sanislo, and M. Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, August 1987.

[7] A.D. Birrell and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[8] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object Structure in the Emerald System. Technical Report 86-04-03, University of Washington, Department of Computer Science, April 1986.

[9] L.A. Clarke, J.C. Wileden, and A.L. Wolf. Object Management Support for Software Development Environments. In *Proc. 1987 Appin Workshop on Persistent Object Stores*, pages 363–381, July 1987.

[10] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88—A Database Programming Language? In *Proc. 2nd International Workshop on Database Programming Languages*, pages 213–229, June 1989.

[11] O. Deux et al. The story of o2. Technical Report 37-89, Altaïr, October 1989.

[12] P.B. Gibbons. A Stub Generator for Multilanguage RPC in Heterogeneous Environments. *IEEE Transactions of Software Engineering*, SE-13(1):77–87, January 1987.

[13] R. Hayes, S.W. Manweiler, and R.D. Schlichting. A Simple System for Constructing Distributed, Mixed-language Programs. *Software—Practice and Experience*, 18(7):641–660, July 1988.

[14] M. Herlihy and B. Liskov. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 4(4):527–551, October 1982.

[15] M.B. Jones, R.F. Rashid, and M.R. Thompson. Matchmaker: An Interface Specification Language for Distributed Processing. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, January 1985.

[16] S.E. Keene. *Object-Oriented Programming in Common LISP: A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[17] Won Kim, Nat Ballou, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. Integrating an object-oriented programming system with a database system. In *OOPSLA Conference Proceedings*, volume 23, no. 11, pages 142–152, San Diego, California, November 1988.

[18] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the Mercury System. Programming Methodology Group Memo 59–1, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, April 1988.

[19] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[20] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an Object-Oriented DBMS. In *OOPSLA Conference Proceedings*, pages 472–482, November 1986. Published as *ACM SIGPLAN Notices*, vol. 21, no. 11, November 1986.

[21] M. Maybee and S.D. Sykes. Q: Towards a Multi-lingual Interprocess Communications Model. Technical report, University of Colorado, Boulder, Colorado, February 1989.

[22] J.H. Morris, M. Satyanarayanan, M.H. Conner, J.H. Howard, D.S.H. Rosenthal, and F.D. Smith. Andrew: A Distributed Personal Computing Environment. *Communications of the ACM*, 29(3):184–201, March 1986.

[23] Patricia A. Oberndorf. The Common Ada Programming Support Environment (APSE) Interface Set (CAIS). *IEEE Transactions on Software Engineering*, SE-14(6):742–748, June 1988.

[24] W. G. Paseman. Architecture of an Integration and Portability Platform. In *Proceedings of the 1988 CompCon*, March 1988.

[25] James M. Purtilo and Pankaj Jalote. An Environment For Prototyping Distributed Applications. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 588–594, June 1989.

[26] Steven P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software*, pages 57–67, July 1990.

[27] W.R. Rosenblatt, J.C. Wileden, and A.L. Wolf. OROS: Toward A Type Model for Software Development Environments. In *OOPSLA Conference Proceedings*, pages 297–304, October 1989. Published as *ACM SIGPLAN Notices*, vol. 24, no. 10, October 1989.

[28] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The Design and Implementation of Ingres. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.

[29] Sun Microsystems, Inc., Mountain View, CA. *External Data Representation Reference Manual*, January 1985.

[30] R.N. Taylor, F.C. Belz, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundations for the Arcadia Environment Architecture. In *Proc. 3rd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 1–13, December 1988. Published as *ACM SIGPLAN Notices*, vol. 24, no. 2, February 1989.

[31] Ian Thomas. PCTE Interfaces: Supporting Tools in Software Engineering Environments. *IEEE Software*, 6(6):15–23, November 1989.

[32] D. Vines and T. King. Gaia: An Object-Oriented Framework for an Ada Environment. In *Proc. 3rd International IEEE Conference on Ada Applications and Environments*, pages 81–92, May 1988.

[33] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. In *Proc. 9th ACM Symposium on Operating System Principles*, pages 49–70, October 1983.

[34] J.C. Wileden and A.L. Wolf. Object Management Technology for Environments: Experiences, Opportunities and Risks. In *Proc. International Workshop on Environments*, September 1989.

[35] J.C. Wileden, A.L. Wolf, C.D. Fisher, and P.L. Tarr. PGraphite: An Experiment in Persistent Typed Object Management for Environments. In *Proc. 3rd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 130–142, December 1988. Published as *ACM SIGPLAN Notices*, vol. 24, no. 2, February 1989.

[36] J.C. Wileden, A.L. Wolf, W.R. Rosenblatt, and P.L. Tarr. *UTM-0: Initial Proposal for a Unified Type Model for Arcadia Environments.* Arcadia Design Document UM–89–01, Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts, February 1989.

[37] Jeannette M. Wing. Writing Larch Interface Language Specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–25, January 1987.

[38] Xerox Corp., Palo Alto, California. *Courier: The Remote Procedure Call Protocol, Technical Report XSIS 038112*, December 1981.

[39] S.B. Zdonik and P. Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proc. 19th Annual Hawaii International Conference on System Sciences*, pages 378–387, 1986.