Talking about Modules and Delivery

Harley Davis Pierre Parquier Nitsan Séniak

Ilog, S.A.
2, avenue Galliéni
94253 Gentilly, France
talk-support@ilog.fr

Abstract

Adding a module system to LISP enhances program security and efficiency, and help the programmer master the complexity of large systems, thus facilitating application delivery. TALK's module system is based on a simple compilation model which takes macros into account and provides a solid basis for automatic module management tools. Higher-level structuring entities — libraries and executables — group modules into deliverable goods. The module system is secure because it validates interfaces, efficient because it separates compilation dependencies from execution dependencies, and useful because it offers a simple processing model, automatic tools, and a graceful transition from development to delivery.

1 Goals

This paper presents TALK's module system, which is simple to understand, smoothly integrates macros, and facilitates application delivery. TALK [5] is a modern LISP dialect which extends the proposed ISLISP standard with a module system, a metaobject protocol, a transparent binding with C and C++[3] and an extensive set of libraries.

The principal goal of TALK's module system is easy and efficient application delivery. Delivery is simplified by providing automatic module dependency tracking tools; delivered applications are made efficient by separating development dependencies from runtime dependencies.

Other LISP module systems see their primary purpose as being namespace control, providing little support for delivery. For example, [8], [2], and [9] allow binding renaming on import, which is a feature used solely for namespace control. In the case of EuLisp [8], it is quite clearly stated that "the module system exists to limit access to objects by name." Similarly, COMMONLISP's packages serve as a large-grained namespace control device. The TALK module system is not primarily geared to managing namespaces, although it does that as well. Instead, it is aimed at large scale program structuring and application delivery, as well as the traditional module system goals of security.

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

In contrast to the header file conventions used by C and C++ programmers and the `defsystem` facility available for many LISPs, the TALK module system is an integrated part of the language. Indeed, we consider it to be the backbone of the language. All language choices must consider the implications for the module system. For instance, unlike COMMONLISP, each defined entity in TALK must have a single point of definition[1]. This allows a static analysis of the contents of a module, and therefore static linking and automatic inter-module depedency analysis.

Unlike the module systems described in [8], [2], and [9], the TALK module system has been driven from the beginning by industrial user feedback. It has been under active development and evolution for over seven years, first in the form of the LE-LISP [1] module system and then in early versions of TALK. The current system has been in active use at industrial sites since April, 1993.

2 Overview

TALK provides a hierarchy of *program units* which map to operating system file types. Each program unit is described by a *description file.* A description file contains a sequence of keys and values describing the various attributes of the program unit. The description file is used by the TALK compiler to construct the object file for each type of program unit.

There are three types of program units:

1. *Modules* contain sequences of definitions and top-level forms. Modules are compiled by TALK's compiler into C source files, which are then compiled by the system's C compiler into standard object files.
2. *Libraries* are collections of modules which constitute a related, coherent subsystem. A library is created by linking together its constituent modules using the system's standard linker into a shared library object file. Because operating systems do not provide nested shared libraries, libraries cannot be nested. In practice, this does not present a problem because libraries are fairly large-grained and tend to be self-contained[2].
3. *Executables* are collections of modules and libraries which constitute a final application. An executable

[1] In CommonLisp, it is not generally possible to statically determine where a generic function is defined.

[2] Other languages which provide nested libraries must implement their own linking and loading solutions for each port. This was deemed undesirable for Talk, which aims for maximum interoperability with standard tools

is created by linking together its constituent modules and libraries (collectively called ***link units***) using the standard system linker into an executable file in the operating system's standard `a.out` format.

3 Compilation Model

3.1 Constructing Compilers

Languages in the Algol family, such as Ada [6] and Modula [4], have had a much easier time introducing modules than languages in the LISP family. Why is this? Algol-type languages are not syntactically extendable. Module systems for these languages need only focus on controlling names of execution entities such as functions and variables. Only one kind of dependency between modules is necessary.

In LISP, the situation is different: LISP has macros written in LISP. Macros are commonly seen as language extensions; a way to introduce new syntactic abstractions. If we take seriously the view that macros are extensions to the language processor, we are led to the conclusion that, in principle, each compilation unit in LISP requires its own compiler. In other words, to compile code in LISP, we must first extend the basic compiler with the macros used by the code, and then compile the code using the new, enhanced compiler[3]. A naive implementation of this model would create an entirely new compiler for each module.

This model is simple. In practice, however, LISP is never implemented in this way. Instead, various mechanisms are used to incrementally and dynamically extend the compiler. In almost all cases, the programmer is responsible for maintaining the compilation environment in a correct state, either by manually loading macros or using mechanisms such as `eval-when` which update the compilation environment during compilation. It is our experience that LISP programmers have a difficult time understanding all the implications of such mechanisms. Even more distressing, these mechanisms must be managed manually because they are too complex to automate. As a result, excess time is spent managing compilation and execution dependencies; interfaces between program units cannot be verified because they are not formalized; and inefficient programs are produced which contain more code and data than they need.

In contrast with these manual mechanisms, each TALK module description file contains a list of other modules (or libraries) used to compile that module, and a list of modules (or libraries) used to execute the module. Module compilation involves first using the compilation modules to construct the correct compilation environment, and then compiling each form in the module. To make the compilation model as simple as possible, no forms in a module are ever evaluated during compilation. No equivalents to the `eval-when` or `macrolet` constructs are provided.

Compared with the compilation model used in most LISPs, TALK's simple processing model is closer to the abstract model described above. The TALK model provides namespace security, since execution references are verified by the compiler; and space-efficient compiled code, since a module's compilation environment is completely separated from its execution environment. A further important benefit of TALK's simple model is that the management of compilation and execution environments can be completely automated, reducing or eliminating the overhead of managing module dependencies. Section 7 describes TALK's analysis tool, which transparently constructs a module's compilation and execution environments.

[3]Indeed, since macros are also written using Lisp code, compiling a macro itself requires the construction of a new compiler Lisp compilation involves a tower of compilers — potentially infinite or circular. In practice, this is avoided because the initial set of macros is coded without macros, or in another language — often an earlier generation of the same Lisp.

3.2 Compilation Process

Compiling a module involves three principal steps:

1. To construct the abstract compiler needed by the module, the unloaded portion of the module's compilation environment is loaded dynamically. This is more efficient than actually constructing a new compiler program. Dynamic loading is explained below.
2. The module's source code is parsed, including macro-expansion.
3. The compiler generates a `.c` file for the module and calls the C compiler to generate a `.o`.

Loading a compiled module also involves three principal steps:

1. The module's execution environment is recursively loaded.
2. The system's dynamic linker loads the module's object file.
3. The module's initialization function is called. This function executes the module's top-level forms and installs its definitions.

Like modules, libraries are also compiled and loaded. Compiling a library means linking the `.o` files of its constituent modules into a shared library object file (usually with a `.so` or `.sl` extension.) Loading a library is the same as loading a module, but a library's initialization function simply calls all of the constituent modules' initialization functions in turn.

Since executables are standalone programs, so they cannot be loaded. Only the compilation is defined, and only one step is involved: Using the system linker, the `.o` files corresponding to the executable's modules and the `.so` files corresponding to the executable's libraries are linked into an `a.out` format executable object file.

4 Modules

4.1 Description

A module is composed of both a source file and a description file. A module source file contains a sequence of definitions and top-level forms which are executed when the module is initialized, either due to dynamic loading or the launching of an executable. Following Ada, this initialization is called *elaboration*.

In addition to some bookkeeping information, module description files use the following keys:

- **export**: The value is a list of defined entities exported by the module.

```
;;; points source file
(defstructure <point> () (x y))

(defun make-fimbulated-point (x y)
  (allowing-fimbulation
    (inverse (make-point x: (fimbulate x)
                         y: (fimbulate y)))))

(defun inverse (p)
  (make-point x: (point-y p)
              y: (point-x p)))

;;; points description file
type:          module
description: "Point module"
package:       myproject
export:        ((class <point>)
                (function make-point point-x point-y
                          make-fimbulated-point)
                (setf point-x point-y))
execution:     (libiltrt fimbulate)
compilation: (libilteval fimbulatemac)
```

Figure 1: The points module: source and description

- execution: The value is a list of other module or library names which export the definitions used to execute the code in the module. This includes any functions called in the module, classes instantiated by the module, and global variables accessed in the module. This list is called the module's *execution environment.*

- compilation: The value is a list of other module or library names which export the definitions which must be loaded to compile the module. This includes any macros and symbolic constants used in the module. Classes can also be required if their slot accessors are to be inlined. This list is called the module's *compilation environment.*

Figure 1 shows a source file and its corresponding module description file. In this example, we create a module named points which defines a structure <point> and some associated functions, including the function make-fimbulated-point. This function uses the macro allowing-fimbulation defined in the module fimbulatemac and the function fimbulate defined in the module fimbulate. We also define an auxiliary function inverse which is internal to the module points.

In the module description file, the module name fimbulate appears in points' execution: key, meaning that this module is needed to execute points but not to compile it. We also see that fimbulatemac appears in the compilation: key. This means that the module fimbulatemac is needed to compile points but not to execute it.

This clear distinction between the execution and compilation environments lets us generate minimal applications; the transitive closure of a module with respect to execution environments gives us all the modules needed for an application. Those modules which are only in the compilation environment are not included in the delivered application. This process is completely natural and does not involve tree-shaking or other manual, error-prone methods.

4.2 Definitions

Each definition in the module, introduced by the special form define, has a name and a type. Defining forms such as defun, defglobal (which defines a global lexical variable), and defclass are macros which expand into define. The definition type determines the binding space in which the entity is bound, and is used to resolve inter-module references. Of the entities defined in a module, a subset can be exported for use in other modules.

TALK does not support per-module namespaces. That is, an exported definition is globally visible. This global visibility is both an advantage and a liability. It is an advantage because it simplifies interactive debugging and on-site application maintenance; these aspects enhance TALK's value as a dynamic language. It is a liability because of the increased potential for namespace pollution and name conflict. As we shall see shortly, this problem is reduced by TALK's package system.

4.3 Macros

As explained above, macros pose a particular problem when designing a LISP module system because they extend the compiler itself and they have access to the full power of the language to compute expansions.

TALK's simple compilation model imposes a restrictive rule on macro use: A macro cannot be used in the same module in which it is defined. More generally, no module can appear in the transitive closure of its own compilation environment. For example, the following code is not allowed within a single module in TALK:

```
(defmacro my-macro ...)

(defun foo () (my-macro ...))
```

The macro my-macro must be placed in another module which is listed in the compilation environment of the module defining the function foo. This restriction also means that forms such as macrolet which require compile-time form evaluation are excluded from the language[4].

Different solutions to this restriction have been suggested, but each breaks down or complicates the simple compilation model which gives the module system its power:

- *Can't you just make two passes when parsing a module to find its macros and pre-load them before the real analysis?*

  The problem with this idea can be illustrated by noting that macro expansion functions could call functions defined in the module itself. Therefore, it is not sufficient to load just the macros; you must load essentially the entire module — before parsing it. This is impossible. Look at the following code:

  ```
  (defmacro foo (arg) (foo-expander arg))
  (defmacro bar (arg) `(list ,arg))
  (defun foo-expander (arg) (bar arg))

  (defun use-foo () (foo ...))
  ```

[4] Nested macros can nevertheless communicate via a special form called dynamic-compiler-let which binds a dynamic variable in the compilation environment to an unevaluated object during the macroexpansion of a set of forms This form poses no problems because it does not require evaluation during compilation

To load this module correctly, we have to rely on the interpreter in the compilation environment as well as the order of definitions in the module. We cannot correctly determine the true compilation and execution environments of the module, and, in fact, either we do not know what to compile or we end up with unneeded code in the final application. In other words, the extension to the compiler ends up being the entire module — and the part of the module solely concerned with extending the compiler ends up in the final program.

Loading a module before compiling it could have undesired side-effects in the compilation environment due to redefinitions, initialisations, and so on. These are the problems that eval-when deals with, but at the cost of breaking down the simple processing model. We prefer a simple, easily explained scheme in which the compilation of a module never involves any execution of the forms in the module. In TALK's system, there is no need for an interpreter in the compilation environment; this also clarifies the processing model for the user.

- *Why not simply forbid local macros from calling functions in the same module?*

  This does not solve the essential problem, which is that allowing macros in the same module in which they are used does not sufficiently distinguish the module's execution environment from its compilation environment. Consider the following code fragment, which does not use any functions at all:

```
(defmacro deffoo ()
  '(defmacro foo () ()))

(deffoo)

(... (foo) ...)
```

  The form (foo) cannot be correctly parsed unless the form (deffoo) is first evaluated. We find ourselves once again in the unenviable situation of relying on (partial) module loading during parsing, and thus complicating the processing model. We consider this solution to be worse than the problem it seeks to correct, and prefer the simpler and more exact solution of simply disallowing the use of a macro in its defining module.

In practice, this restriction poses few hardships. First, most systems have relatively few macros compared to the number of functions. Second, by placing macros outside of the modules in which they are used, the size of the final application is reduced. Third, small macros which are not syntactic extensions could be replaced by inlined functions, which do not pose the same problems since their "expansion" code is completely transparent to the compiler: They do not require any language extension. This is why inlined functions are acceptable in Algol-like languages such as C++, while full-powered LISP-style macros are not.

Some people have worried that this restriction will lead to an infinite regress of module levels: Macros which generate macros need an extra level of modularity, and so on. It is our perception that systems this complex will benefit from the structuring imposed by the module system, and they will be complicated enough that modularizing the code will hardly be the major problem.

```
type:        library
description: "Fimbulated point system"
modules:     (points fimbulate)
export:      ((points
                (class <point>)
                (function make-point point-x point-y
                          make-fimbulated-point)
                (setf point-x point-y))
              (fimbulate
                (function fimbulate)))
execution:   (libiltrt)
```

Figure 2: The libfim library description file

5 Libraries

5.1 Description

Libraries are collections of modules. They are implemented as shared libraries, which means that multiple applications using the library on a single machine share code in memory. In addition, most operating systems provide rapid dynamic linking facilities for shared libraries.

Like modules, libraries must be elaborated before they can be used. Elaborating a library consists of elaborating each of its constituent modules in a predefined order.

A library description has the following primary keys:

- modules: The value of this key is an ordered list of module names which constitute the library. The order of modules in this list determines the order in which the modules will be elaborated when the library is loaded or initialized as part of an executable.
- export: The value of this key is simply a union of the exports of each of the constituent modules. It is needed to deliver the library separately from its modules.
- execution: The value of this key is the union of the values of the execution: key of each of the link units, removing the link units themselves.

Figure 2 shows a sample library which contains two of the modules from our previous example: points and fimbulate. Together, these two modules form a complete, deliverable point fimbulation subsystem which some enterprising young cad hopes to sell to the masses.

5.2 Namespaces and Packages

Because TALK uses global namespaces for each type of entity, a library's exports must be the exact union of the exports of its modules. No additional name hiding can be done at the library level. However, often a module exports an internal function to be used only by other modules in the library.

For example, module m1 defines and exports the following unsafe, internal function:

```
(defun unsafe-foo (x) ...)
```

Module m2 defines a safe, documented version of foo:

```
(defun foo (x)
  (assert ...)
  (unsafe-foo x))
```

Unfortunately, unsafe-foo, as an exported function, has global visibility and dangerously pollutes the global function namespace. To solve this problem, TALK has a package system which is entirely orthogonal to the module system.

A TALK symbol is composed of both a name and a package. Syntactically, a fully packaged symbol is prefixed by its package and a dot. Additionally, each module has a package into which symbols beginning with just a dot are read. For example, the symbol foo.bar has the package foo and the name bar. The same symbol appearing in a module specifying foo as its read package could simply be written as .bar. The symbol bar without a dot is an unpackaged symbol.

Extensive namespace pollution can be easily avoided by using a simple packaging convention, supported by the programming environment:

- Each project defines a unique package used by all of its modules.
- Unpackaged symbols form the interface to the project and are documented.
- Packaged symbols are internal to the project and are not documented.

Using this system, our problematic case could simply be treated by naming our troublesome function .unsafe-foo rather than unsafe-foo. Users of this project would be forewarned that they should not use symbols in the project's package.

This simplistic solution will not satisfy purists who insist on complete hiding and absolute security. However, it has important advantages. First, it is simple to implement and understand. Second, by providing access to undocumented definitions, debugging and on-site maintenance are greatly simplified. These are important benefits of a dynamic language; we should be loathe to give them up in LISP.

6 Executables

6.1 Description

An executable is a deliverable program unit representing a turnkey application. The TALK development environment itself is delivered as an executable, and programs developed by users are also delivered as executables.

An executable contains an ordered list of modules and libraries which are elaborated to initialize the application. The last module typically contains a top-level form which runs the application by starting a top-level loop or performing a calculation and leaving. This module corresponds to the main function in a C program.

Figure 3 shows a typical executable description file for an executable which includes the libfim fimbulated point library described above. When built, this executable initializes the libraries libiltrt (which implements the basic TALK datatypes), libilteval (which includes the interpreter and the basic TALK macros), libfim, and the standard TALK module iltrune. This standard module starts up a top-level loop, so the interpreter library is needed.

```
type:        executable
description: "The fimbulated point application"
units:       (libiltrt libilteval libfim iltrune)
```

Figure 3: The fimexec executable description file

Closed applications do not need the interpreter, substantially reducing their size. Although this application includes an interpreter, it does not include the compiler and development environment; no module in this application has an execution dependency on them, so they are naturally excluded by never being included in the first place.

7 Analysis Tool

7.1 Overview

The module system as described above is a complete, functional system. However, because it involves the management of description files which are tedious to maintain, it is not by itself a practical system for medium or large sized applications. Indeed, if the programmer must manually maintain these files, all of the advantages accrued by the system's structure are lost because the development cycle is much slower than with current practice.

Therefore, TALK also includes an analysis tool to automatically and transparently maintain description files for the programmer. Because TALK's basic module system is simple, straightforward, and complete, the analysis operation provides guaranteed safety and efficiency: If no interface mismatches are detected by the analysis tool, the final application will not contain any missing implementation components. In addition, the analysis tool will automatically create a nearly minimal application, in which the compilation and development environment, including the compiler itself, are not needed.

Because the analysis tool is not strictly necessary for the overall functioning of the system, it is considered an environment feature rather than part of the language. Nevertheless, in practice users only work with program units via this tool, and rarely, if ever, manually modify the actual description files. We have found that the analysis tool not only eliminates the structural burden imposed by the module system, but, compared to other LISP systems, actually speeds development, especially near the end of the development cycle when delivery preparation begins. In fact, because the program uses the formal structures of the module system from the beginning of development, final delivery is almost immediate. This painless delivery is the ultimate goal of TALK's module system.

To transparently maintain the description files, we define an analysis operation which is applicable to each type of program unit. The implementation of this operation is linked to the compiler by a set of standard conditions signaled by the compiler indicating the references detected in the module. The analysis tool provides condition handlers which find the correct exporting program units and update the program unit description files accordingly. By relying on the condition system to transmit information between the compiler and the analysis tool, TALK maintains a clean separation between the basic language processor and the environment in which the language is used. This allows the environment to be replaced and updated independently from the language as new ideas and methods for the environment are developed.

7.2 Module Analysis

Initially, the user need only write the source code for a module. The first time the module is analyzed, a description file is created containing the appropriate initial information.

The analysis operation for a module performs the following steps:

1. The compiler makes a pass over the module source code, detecting and signaling all external definition references, including function calls, macros used, classes referenced (via method definition or subclassing, for example). The compiler also signals all the entities defined in the module and whether they are used locally.
2. The analysis tool looks for program units exporting unresolved references. Any definitions not found are resignaled with the source code locations of the references.
3. Based on this information, the analysis tool updates the module's execution and compilation environment. In addition, the analysis tool updates the module's exports by exporting any unused or unpackaged definitions from the module and removing any undefined exports.

7.3 Library Analysis

Initially, the user need only specify the modules in a library. The analysis tool does everything else.

A library does not contain source code, so the analysis of a library does not call upon the compiler. However, the following steps are taken to ensure that the library's description file is correct:

1. The library's execution environment and exports are computed by taking the union of the execution environment and exports of each of the library's modules.
2. The modules are sorted using a stable topological sort based on the *in-execution-environment-of* relation to find a valid initialization order for the library.

7.4 Executable Analysis

To create an executable, the user need only indicate the startup module and any of the project's libraries and modules not referenced directly by the startup module[5]. Everything else is computed automatically.

The analysis of an executable involves the following steps:

1. The analysis tool computes the transitive closure of the executable's link units with respect to their execution environments to find all of the modules and libraries which must be linked with the executable.
2. The executable's program units are sorted using a stable topological sort to find a valid initialization order.

[5]This is necessary if the startup module just starts a toplevel loop. In completely closed applications, the startup module typically references the entire project via chains of function calls

8 Other Issues

8.1 Method References

Generic function references pose a problem for the analysis tool. When a generic function is called in a module, it is easy to detect and install a reference to the generic function definition itself. However, the methods for the generic function may be spread out over several modules. Consider the following code in four separate modules:

```
In module m1: (defgeneric foo (x))
In module m2: (defmethod foo ((x <list>)) ...)
In module m3: (defmethod foo ((x <string>)) ...)
In module m4: ... (foo x) ...
```

It is clear that the module m4 needs to import m1. However, it is not clear which, if any, of the modules m2 and m3 are needed. If both of these methods were imported into m4, the application might end up including many spurious modules. The problem becomes more serious when we consider very general generic functions such as `prin-object` or `initialize-instance` for which methods may be defined for nearly every class.

Instead of using complicated and unstable type inferencing, or some other solution difficult to implement and understand, TALK adopts a practical approach to this problem which has proven more than sufficient in practice: Users are encouraged to follow a simple convention in the placement of their method definitions. Methods should be defined either in the same module which defines the generic function, or in the module which defines one of the discriminating classes for the method. In this way, a static reference to the generic function or the discriminating class will correctly ensure that the appropriate method is imported. Generic functions are statically referenced by calling them or referring to them using the special form `function`. Classes are statically referenced either by using them as superclasses, specializers, or by calling one of their constructors or accessors, or by referring to them using the special form `class`.

For other cases of dynamic or invisible references, features can be defined and statically referenced using the macros `deffeature` and `feature`. The use of features allows the analysis tool to correctly generate an execution environment.

8.2 Elaboration Order

The elaboration order problem is that of finding a correct order of initialization for modules in a library or executable. The problem arises because TALK allows modules to contain top-level forms and cross-references for execution. In fact, this problem is not limited to LISP; see [7] for a discussion of the elaboration problem problem in Ada. The following code fragment in two modules illustrates the problem:

In module m1:

```
(defun foo () (bar))
```

In module m2:

```
(defun bar () ...)

(foo)
```

In this code fragment, the analysis tool detects a reference from m1 to m2, and from m2 to m1; each module appears in the other's execution environment. Based purely on this information, no unique initialization order can be

Unit	Type	Code Size
libiltcrt.so	library	229,376
libiltrt.so	library	835,584
libilteval.so	library	811,008
libiltdev.so	library	868,352
italk	executable	40,960
libc.so	system library	442,368

Figure 4: Sizes of standard TALK program units and libc for comparison on Sun OS 4.1

determined between m1 and m2. However, only the order (m1, m2) is valid because foo must be installed before it is called. If m1 and m2 are in the same library or executable, a simple topological sort will not always come up with the right answer.

For this reason, TALK uses a stable topological sort in the analysis of a library or executable. The analysis tools detects and signals any newly-discovered cycles in the execution dependencies. The programmer corrects the error by switching the order of modules in the library or executable description. Due to the stability of the topological sort, subsequent analyses will preserve this order to break such cycles.

Alternatives to this simple approach include distinguishing between execution dependencies and elaboration dependencies. However, determining the exact, minimal set of elaboration dependencies is equivalent to the halting problem. Instead of adopting a complex, heuristic approach, we decided to go with the simple solution described above, and see how bad the problem was in practice. Since the introduction of this system in April of 1993, none of our users has yet run across the problem in practice.

9 Efficiency

By separating a module's dependencies into distinct compilation and execution environments, and by using standard shared library technology, TALK's module system reduces final runtime requirements. TALK itself is delivered as four libraries:

- libiltcrt: This library contains the core TALK runtime features such as the memory manager.
- libiltrt: This library contains the basic TALK runtime modules including the fundamental datatypes and the object system.
- libilteval: This library contains the interpreter and the basic TALK compilation modules.
- libiltdev: This library contains the compiler, the analysis tool, and the other development environment tools.

Figure 4 shows the code sizes of these libraries for SunOS 4.1, as well as the size of the basic TALK executable italk. For comparison, it also shows the code size of the standard C runtime library libc, which includes very few datatypes and no interpreter or development environment. The comparable TALK libraries, libiltcrt and libiltrt, are together a little more than twice as large as the C library.

10 Conclusion

The design of the TALK module system described here has been heavily influenced by user feedback from earlier versions. Previous versions required manual management of module interface files; this was deemed too unwieldy by users and slowed development time. Furthermore, previous versions had complex sets of inter-module dependencies to manage module interaction at several levels. These controls, while allowing superior optimization potential, were found to be confusing and difficult to master. Simplicity was deemed more important than fine-grained control. In response to this feedback, the new module system allows completely automatic module management and only distinguishes the two essential inter-module dependency links: compilation time versus execution time. User feedback for the current version has been uniformly positive; users derive the benefits of the module system without sacrificing rapid development or efficiency. We believe that automation and simplicity are the keys to the success of TALK's module system.

The TALK module system is based on a simple module compilation model which allows us to build up higher level functionality step by step, each step relying on the solidity of the step below. The analysis tool provides automatic, transparent security by validating and maintaining inter-module dependency links. The system provides efficiency by clearly distinguishing between compilation and execution dependencies; compilation units are simply not included in deliverable libraries and executables because there are no runtime references to compilation units in runtime units. Higher-level structuring units let us provide deliverable goods which can be constructed very simply. The use of standard operating system tools and file types means that these deliverable goods are easily integrated into the final operating environment and automatically benefit from improvements in the basic tools.

References

[1] Jérôme Chailloux, Mattieu Devin, and Jean-Marie Hullot. Le-Lisp: A portable and efficient Lisp system. In *ACM Conference on Lisp and Functional Programming*, pages 113–122. ACM SIGPLAN, ACM Press, 1984.

[2] Pavel Curtis and James Rauen. A Module System for Scheme. In *ACM Conference on Lisp and Functional Programming*, pages 13–19. ACM SIGPLAN, ACM Press, June 1990.

[3] Harley Davis, Pierre Parquier, and Nitsan Séniak. Sweet Harmony: The Talk/C++ Connection. In *ACM Conference on Lisp and Functional Programming.* ACM SIGPLAN, ACM Press, 1994.

[4] Luca Cardelli *et al.* Modula-3 Report (revised). Technical Report 52, Digital Equipment Corporation Systems Research Center, 1989.

[5] Ilog. *Ilog Talk Reference Manual*, 1994.

[6] American National Standards Institute, Inc. The programming langugage Ada reference manual. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science.* Springer-Verlag, 1983.

[7] Leslie C. Lander, Sandeep Mitra, Nitin Singhvi, and Thomas F. Piatowski. The Elaboration Order Problem of Ada. *Software Practice and Experience*, 22(5):391–417, 1992.

[8] Padget, J.A., Nuyens, G., and Bretthauer, H. An overview of EuLisp. *Lisp and Symbolic Computation*, 6(1/2):9–98, 1993.

[9] Sho-Huan Simon Tung. Interactive Modular Programming in Scheme. In *ACM Conference on Lisp and Functional Programming*, pages 86–95. ACM SIGPLAN, ACM Press, 1992.