An Extensible Programming Environment for Modula-3*

Mick Jordan†

Abstract

This paper describes the design and implementation of a practical programming environment for the Modula-3 programming language. The environment is organised around an extensible intermediate representation of programs and makes extensive use of reusable components. The environment is implemented in Modula-3 and exploits some of the novel features of the language.

1 Introduction

The success of a programming language owes much to its associated programming environment, especially to the set of tools which enable the construction and modification of programs. With a new language there is trade-off between producing an implementation quickly and in providing a rich and powerful set of tools. Choosing the first approach typically results in low quality, poorly integrated tools. The second approach can result in an unacceptable delay before the language can be used. This paper describes the design and implementation of a toolset for Modula-3 [Car89], a new programming language that adds threads, objects, exceptions and garbage collection to Modula-2 [Wir83]. Our aim was to steer between the two extremes noted above and produce a basic set of tools designed around an open, extensible framework, to which new tools could be added incrementally.

*This work was carried out at Olivetti Software Technology Laboratory, Menlo Park, CA 94025, USA.

†Author's current address: Digital Systems Research Center, 130 Lytton Ave, Palo Alto, CA 94301, USA

2 Goals

The long term objective was to build a practical environment to support large scale software development in Modula-3. Initially, the environment would contain only a basic set of well integrated tools such as a compiler, linker and debugger, but would be capable of supporting real projects on a variety of systems. In the longer term additional tools such as code browsers, code analysers, and improvements like smart recompilation would be added, building on the basic environment. In this paper, we are more concerned with how tools are constructed and combined than with the exact details of their functionality or interface to users of the environment. The idea is that, if the construction of new tools is made easy enough, new and powerful tools will appear routinely. Important sub-goals for the environment were as follows:

A Framework for Program Analysis and Transformation

There is a large class of tools which engage in program analysis or transformation, for example, browsers, compilers, pretty-printers. These tools require access to an intermediate representation of program source, and support mechanisms to facilitate the analysis. It is important not to prejudice the addition of such tools by choosing an overly restrictive framework.

Extensibility

In order to make effective use of the framework, adding a new tool must be straightforward. Detailed knowledge of other tools should not be necessary. New tools should have equal status with the initial set. Note that the extensibility is aimed at tool developers rather than at users of the environment, although the design does not preclude the latter as a subsequent enhancement.

Portability and Availability

The system must be portable to a wide range of systems and execute on stock hardware, typically 3 Mip workstations with 8-16 Mb of main memory. This placed some constraints on the system design and on the degree to which we could incorporate external tools.

Implement in Modula-3

The implementation of a programming environment extends from low-level (unsafe) programming through to application-level programming. This is precisely the area to which Modula-3 is targeted and we believe that Modula-3 offers a better set of facilities for this area than its competitors. Implementing the compiler for a (systems) language in itself is also a good baseline test of an implementation, and lends credibility to a new language.

3 Design

The two key features needed to achieve the goals are:

- A well defined and extensible intermediate representation of program text.
- A system design that encourages the reuse of components.

The foremost requirement of the intermediate representation (IR) is that it be information preserving. That is, with the possible exception of lexical details, all information present in the source text must be represented in the IR. The second requirement is that the IR be capable of recording additional information, perhaps implicit in the source text, that is generated by other tools in the environment. In short the IR must be extensible. One such IR that satisfies these requirements is an attributed Abstract Syntax Tree (AST), which is well established in the literature as a sound basis for compiler development [Aho86, Bor88, Don84, Goo83, Rep89]. An extensible AST provides an excellent integration mechanism for the toolset. However, for extensibility to really work well, new tool developers must be able to compose existing tools in order to generate suitable AST instances for their task. They must also be provided with a rich set of packages or tool fragments that can be invoked to simplify their task.

4 The Modula-3 Abstract Syntax Tree

4.1 Background

A detailed discussion of the merits of an AST as the intermediate representation for programs is beyond the scope of this paper. The DIANA reference manual [Goo83], which defines an AST for Ada[1] [DOD83], contains an excellent analysis of the fundamental issues and has many useful recommendations for AST designers. Indeed the design of the Modula-3 AST (M3AST) was strongly influenced by DIANA and several techniques and conventions were adopted directly. Where M3AST differs from DIANA is in the explicit use of an object-type hierarchy to capture commonality and in the separation of the specification into pieces that correspond to individual tool fragments. Also the notion of M3AST as the output from a compiler front-end is de-emphasised in favour of support for a wider collection of tools. These techniques contribute to the readability and maintainability of the specification and aid in supporting extensible tool development. To illustrate the confusion that can arise with the monolithic approach consider our previous experience with an AST for Modula-2+. This AST was defined in a single interface as a collection of variant record types. It was rather complex, and it was not always clear *when* a given node or attribute was set, by *which* phase (or tool) and whether an attribute was *temporary* or (usefully) *persistent*. This confusion did not facilitate extensibility, which demands some form of incrementality in the specification.

4.2 AST Structure

The AST is modeled as an attributed tree. Following the terminology used in DIANA, the tree contains named *nodes* that correspond to non-terminals in an abstract version of the Modula-3 grammar. Members of the right-hand side of the production rule for a non-terminal are denoted by typed attributes of the node. Non-terminals with alternative productions, for example, `Expression`, are modeled as *classes* whose members are the associated set of nodes. In other systems, for example the Synthesiser Generator [Rep89], classes are referred to as *phlya* and the subtree rooted at a node is called a *term*. Attribute types may be basic types such as `INTEGER`, user-specified abstract types, such as might be used to denote an identifier name, or a node or class type. The repetitive constructs in grammar productions are represented by an attribute of type *sequence*. The set of nodes, classes and attributes corresponding to the language syntax form the backbone of the extensible AST and comprise the lexical and syntactic *views*. A new view is created by adding extra attributes to existing nodes and perhaps by defining new node and class types. An example is the semantic view, where attributes to denote identifier bindings are introduced. The views form a hierarchy that initially reflects the traditional sequence of phases in a compiler. An important kind of view is one which contains nodes and classes that represent an embedded language with its own syntax and

[1] Ada is a registered trademark of the U.S. Government

semantics, for example a specification language such as Larch [Gut85]. The ability to share a single formalism and to establish interconnects between the two ASTs is very useful. In the basic toolset there are the following AST views:

Lexical This view is very simple, defining only the basic types needed to denote identifiers, numbers, comments etc. These are used as attribute types in the syntactic view.

Syntactic The syntactic view defines the backbone of the AST, which other views decorate with additional attributes. There is a very close relationship between the syntactic view and the Modula-3 grammar.

Pragmas The revised language provides a pragma mechanism, in the style of comments, which is sufficiently general to support small-scale language extensions or embedded languages. The pragma view defines new nodes to represent the standard set of pragmas and adds additional attributes to the appropriate nodes in the syntactic view.

Semantic This view adds the additional attributes that result from semantic analysis, for example, denoting the binding between a used identifier and its defining occurrence. At this level the AST becomes a graph rather than a tree.

Semantic Temporaries This view factors out those attributes that are used during semantic analysis but that have no (interesting) long term value, or are too costly to maintain in persistent form. One role of this view is to provide attributes that avoid the source-level bias of the syntactic view, for example, normalising identifier declarations within a block and adding symbol tables that provide fast lookup. The lack of such features in DIANA has been criticised by compiler designers [Zor85].

Front-End This view defines attributes to record the status of the compilation, the mapping between an AST instance and an external file, and a mechanism for separating connected ASTs into individual instances.

Inline Procedure inlining is implemented as an independent tool. Some additional nodes are defined to support the process, for example, a node to describe an inlined function procedure in an expression.

C-Code Generator This view defines attributes used during the code generation phase. Most of these attributes are temporary but some are exported to other views, for example, information on the initialisation requirements of a module.

Pre-Linker This view supports the pre-linker tool, described in section 5.4, and defines attributes of a more global nature, for example the *depends-on* relation for a module, which is used in determining program initialisation order.

4.3 Mapping the AST into a Programming Language

The above discussion treats the AST in an abstract manner, but ultimately this abstraction must be mapped into some programming language. This can be achieved in many ways and involves several classic space/time/flexibility trade-offs. In making these decisions we believe that it is very important to maintain the strongly-typed nature of the abstract AST. This is easy to achieve for any fixed set of AST views, but it should also hold for additional views, in order that new tools added by extension have equal status and protection. The key issue is how to support the addition of new attributes on existing node types. There are essentially two approaches. Either augment the AST directly with new attributes or construct auxiliary data structures which refer back to the AST as necessary. The latter approach is tedious, inconsistent with using the AST as a tool integration mechanism, and can waste space because structural aspects of the AST, for example block structure, may have to be repeated. So we are led to direct augmentation. Unfortunately, the ability to define aggregate data-types incrementally is typically not found in strongly-typed programming languages. The Interface Definition Language (IDL) [Sno89], used to define DIANA, supports this feature, and therefore the view model, through structure *derivation*. A new (data) structure, or view, is derived from one or more previous structures by specifying additional new attributes, nodes and classes. Since the information about any one node may be scattered among several specifications, global knowledge is required to compile a client of any single structure. IDL is further disadvantaged by containing no operational capability. It contains a powerful assertion language but the generation and manipulation of the structures must be carried out in a separate programming language. Not suprisingly much of the benefit of the rich type system is lost when it is mapped into a language like C [Ker78] or Ada. An obvious and traditional approach to attribute storage is to attach a property list to every node and resort to run-time type-checking on attribute access. This is very flexible but it is wasteful of space, much less efficient than direct access and considerably more tedious to program (thus leading to less maintainable code). Fortunately, the object types available in Modula-3 offer an alternative solution, and this is discussed in section 4.5.

4.4 An Overview of Object Types in Modula-3

An understanding of Modula-3 object types is helpful in understanding the next section. Object types in Modula-3 are similar to those found in other strongly-typed, object-oriented languages, such as C++ [Str86], Eiffel [Mey88] and Object Pascal [Tes85]. However, the notions of *interface* and *class* are kept distinct. An object type is simply another kind of type-constructor and is similar to a reference to a record of data fields, except that an object type may contain methods and may also inherit fields and methods from a statically designated supertype (ancestor). Methods encapsulate the operations that are permitted on an object; a method always receives the invoking object as its first parameter. Modula-3 supports information hiding through *opaque* objects and also permits different degrees of opacity by a mechanism called a *revelation.* For example:

```
TYPE
  ST = OBJECT s: INTEGER END;
  T = ST OBJECT t: INTEGER END;
  U <: T;
```

The type `T` is an object with field '`t`' plus those fields inherited from `ST`, which has no ancestor. `T` is a concrete type and an instance can be created by a call to `NEW(T)`. The type `U` is an opaque object type, introduced by the subtype operator '`<:`'. The interpretation of this declaration is that, in its concrete form, `U` will be some subtype of `T`, that is, it will have the fields and methods of `T` and perhaps additional ones. Typically these will be the private data needed to support the abstraction that `U` represents. A client of the above declaration for `U` cannot create an instance with `NEW`; this is only possible in the scope of a concrete definition of `U` [2]. To distinguish a declaration of a new type from its concrete definition, Modula-3 uses a *revelation*, thus:

```
REVEAL U = T BRANDED OBJECT u: INTEGER END;
```

The `BRANDED` keyword ensures that the opaque type is unique and is required because Modula-3 compares types using structural equivalence. A variation of `REVEAL` permits information on an opaque object to be revealed in differing amounts to different clients. For example:

```
REVEAL U <: T OBJECT u: INTEGER END;
```

This provides as much information as the previous revelation, but does not preclude `U` from containing yet further fields and methods. All revelations are checked for consistency before a program is executed.

[2] The interface declaring `U` will typically export a procedure to create an instance.

4.5 The AST in Modula-3

The AST is specified as a collection of interfaces based around the views described in section 4.2. Nodes and classes are both represented as object types. All node types are members (subtypes) of a given class. An AST instance consists of a collection of connected nodes; there are never any instances of class types. To make the distinction between a class and a node clear we follow the DIANA convention and use all capitals for class names. The base interface, `M3AST`, defines a single class, `NODE`, of which all nodes in the AST are subtypes. The interface `M3AST_AS` defines the collection of classes and nodes in the syntactic view. At this level of abstraction only the subtype relationship is revealed, e.g. `Integer_literal <: EXP <: NODE`. The attributes of a node or class, and the way in which they are accessed, is left to other interfaces, in order to allow flexibility in representation. One might choose to expose the attributes as fields of an object, or to require access via methods, or even to use a conventional procedural interface. To date we have stuck with fields, preferring to trade AST size for speed of access [3]. The interface `M3AST_AS_F` defines the view that provides node and class attributes as object fields. The technique for defining a view fragment is to define a local type that contains the attributes and then to reveal that the abstract type is some subtype of this fragment. For example:

```
INTERFACE M3AST_AS_F;
IMPORT M3AST, M3AST_AS;

TYPE UNIT = M3AST.NODE OBJECT
    as_id: M3AST_AS.UNIT_ID;  (* etc. *)
  END;
REVEAL M3AST_AS.UNIT <: UNIT;
```

The `UNIT` class contains two members, `Interface` and `Module`, which differ only slighly. Many of the attributes are common to both members and are thus attached to the class and inherited. If a subsequent view wishes to add additional attributes to a `UNIT` [4], it does so in a similar way. The example below is an extract from the C-code generator view, which adds an attribute to record whether a unit requires specific initialisation on program start-up. This is an example of an attribute that is exported to another tool, in this case the program linker.

```
INTERFACE M3AST_CG_F;
IMPORT M3AST_AS, M3AST_AS_F;
```

[3] Representing node references as, say, small indices into a node table, is one way to reduce the size of the AST.

[4] Our convention is to reuse the name of the abstract type for the fragment. Since Modula-3 supports qualified naming, which we use consistently, no confusion should result.

```
TYPE UNIT = M3AST_AS_F.UNIT OBJECT
    cg_init_status: INTEGER
  END;
REVEAL M3AST_AS.UNIT <: UNIT;
```

The supertype of a view fragment is always either the true parent of the abstract type, as specified in the initial subtype hierarchy, or a fragment in a previous view. Modula-3 forces this linearisation of the view hierarchy because it insists that all revelations for a type be consistent at compile time, rather than at link time. Thus, when defining a new view, one must know which view last added attributes to the node. Fortunately this information is contained in the interface that binds all the views together. This interface makes concrete revelations for all the abstract node and class types in the system, for example:

```
INTERFACE M3AST_all;
IMPORT M3AST, M3AST_AS;
IMPORT M3AST_AS_F, M3AST_CG_F;

REVEAL M3AST.UNIT =
  M3AST_CG_F.UNIT BRANDED OBJECT END;
(* and so on for all nodes and classes in
   all views to be included. *)
```

Aside from the nuisance of the enforced linearisation, which has not proved a problem in practice, the above mechanism has three important features, which contribute greatly to extensibility:

- Code compiled against a given set of views need not be recompiled when a new view is added.
- Only the attributes in explicitly imported views are visible to a tool [5].
- It is possible to build tools that contain a subset of the views, subject to the constraints imposed by the view linearisation. For example, one can build a compiler front-end that uses an AST without the code-generator attributes. All that is required is to edit the configuration file, recompile it and relink the tool.

There are some disadvantages which can cause performance problems:

- The real type hierarchy is made artificially deeper by the view supertypes.
- The compiled code for attribute (field) access must use link-time addressing.

[5] Attributes in view supertypes are invisible since Modula-3 does not propagate revelations through more than one level of import.

Both these issues can be resolved by appropriate global optimisations, since the problems only result from compiling with partial knowledge. Indeed, one tool that we would like to add to the environment is such a global optimiser. An alternative approach is to exploit the environment and build a tool that automatically generates a single monolithic interface from the views, by flattening out the view supertype list into a single object type. We have implemented this tool and by virtue of the reusable components described in section 5, it is only 150 lines of Modula-3. Using this approach results in a development environment rather similar to that afforded by IDL, where global recompilation is necessary after a change to any view.

4.6 The Cost of Extensibility

Node attributes are either basic (abstract) types or references to other nodes and classes (object types). The extensibility of the AST is based on using object types wherever possible, since only these types can be subtyped to add new attributes. It is tempting to immediately optimise some AST classes into basic types; for example binary operators can be represented as an enumeration type. This is tempting because, on stock workstations, the large size of ASTs remains a concern. However, such optimisations hinder extensibility and may force tools to construct more complex auxiliary data structures. Extensibility has a price but we believe that premature optimisation is wrong. Our experience is that new tools typically attach attributes to a small subset of the existing node types, and this only results in a small percentage increase in the size of the AST. To understand this issue better, we are carrying out measurements of the ASTs that result in practice. One observation so far is that AST sequences tend to be rather short, averaging slightly less that two members, with about half of all sequences having length one. This information can be used to pick a sequence representation that reduces the overall space requirements [Jor90].

4.7 Persistent ASTs

One purpose of an extensible AST is to allow one tool to compute and store information for use by another tool. With current computer workstations, it is not always practical or desirable to integrate all the tools into a single program. For example, the ASTs for the Modula-3 compiler itself occupy about 16Mb of virtual memory. Hence, there needs to be a mechanism for making an AST persistent. There are other reasons, such as precompiling frequently used interfaces for direct loading in AST form. The general problem of making complex data structures persistent has been addressed by several groups, e.g.

[Her82], [Neu88], often in the context of remote procedure call. IDL also provides powerful mechanisms that include support for automatic data type translation, although we found these to be rather inflexible.

The term *pickling* [Bir87] has been coined to define the transfer of data between an executing program and an external store, in a way that preserves type safety and storage manager invariants. Given sufficient type information at run-time, a pickling system can be implemented that requires no extra effort on the part of an application programmer. However, it can also be implemented by a tool which, operating from the AST, generates stub code to process an explicit set of data structures. Since this is a compile-time solution, its performance can be better than the run-time version. We chose the stub-based approach and our design and implementation is based on a thorough study of the issue, especially relating to object types, by Craft [Cra89] for C++.

Pickling ASTs is complicated by their highly interconnected structure, which results from the import of one unit by another in the source text. Naively pickling an AST will save the entire connected set of ASTs, resulting in massive duplication in the persistent store. Our solution to this problem is similar to others [Har89], [Wil88], involving surrogate nodes, but the interaction with automated pickling is new. First, by design, we restrict external references to just two classes of nodes in the AST, defining-identifiers and type-specifications. This occasionally causes an extra level of attribute indirection but overall provides significant simplification. We replace references to external nodes by surrogate nodes, that are new subtypes of these classes, which remember how to locate the external node. The generation of the surrogate nodes is done on the fly as the AST is pickled and so is non-destructive. This kind of dynamic intervention is impossible using the static transformation mechanisms provided by IDL. The recombination of the ASTs is integrated with the consistent compilation checking of Modula-3, which ensures that mismatched ASTs are not erroneously recombined. In our present system unpickling is not handled lazily - to do so would require a residency check on every access to an attribute of the external classes, and this would demand a procedural style of access.

5 Reusable Components

5.1 Introduction

The ability for one tool to reuse components that were developed initially for other tools is crucial to the extensibility of the environment. It is also a difficult engineering problem requiring discipline and the resolve to redesign and restructure existing components if foresight was lacking. Reusability is encouraged by focussing a component on a small, well-defined task, by clearly specifying the pre and post conditions and by avoiding the use of static or global state. The inherent complexity of many programming environment components makes this a tall order. The extensible AST is of significant help since it is often possible to capture the inputs, outputs and temporary state directly in the AST. The design of a reusable component is made much harder if explicit storage deallocation is required, so using a language, like Modula-3, which provides garbage collection, is an advantage.

5.2 Packages and Tools

The environment components divide into two groups: *packages* such as the tree walker described below, and *tools* which carry out some action on behalf of a user, for example a browser. A tool is akin to a UNIX[6] process, but is integrated with other tools by the AST and auxiliary data structures, rather than by pipes and files. Since the AST can be made persistent, tools can be built stand-alone or bundled together.

5.3 Packages

The environment provides a collection of packages for managing files, the user-interface, the AST, and so on, which are themselves built on an application independent Modula-3 library. We will briefly discuss the four packages that a new tool developer is most likely to use.

The Tree Walker

The most important building block of the environment is the tree walker which forms the engine-room of all the syntax directed tools. The purpose of the tree walker is to factor out the structure of the AST, and how it is traversed, from the real job of the tool. The tree walker is object oriented; its main routine applies the abstract *walk* method associated with a node, which is implemented by a module in the view that declares the node type. The tree walker provides a callback mechanism to the client tool allowing control to be gained as each node is visited, either on entry to the node, on exit, or both. The walk order is determined by the implementation of the walk method; for the syntactic view it is the obvious preorder traversal. In the callback there is almost always a need to access non-local state. Modula-3 does not provide full closures, but auxiliary state can be associated with a procedure activation using object types as shown below. Here is an excerpt from the tree-walk interface and an example client which is interested in finding procedure call sites.

[6] UNIX is a trademark of AT&T Bell Laboratories.

```
INTERFACE M3ASTWalk;
TYPE
  VisitMode = {Entry, Exit};
  VisitModeControl = SET Of VisitMode;
  Closure <: OBJECT METHODS
    callback(n: M3AST.NODE;
             vm := VisitMode.Entry);
  END;

PROCEDURE VisitNodes(
    n: M3AST.NODE;
    vc: Closure);
(* Walk tree rooted at 'n', applying method
   'vc.callback' on entry to each node. The
   heart of this procedure is a single call
   to 'n.walk()'.
*)
END M3ASTWalk.

MODULE FindCalls;
IMPORT M3AST, M3AST_AS, M3ASTWalk;
IMPORT IO, StdIO;

TYPE
  StreamClosure = M3ASTWalk.Closure OBJECT
    s: IO.Stream;
  METHODS
    callback := VisitCalls
  END;

PROCEDURE VisitCalls(cl: StreamClosure;
   n: M3AST.NODE; vm: M3ASTWalk.VisitMode);
  BEGIN
    TYPECASE n OF
    | M3AST_AS.Call(call) => (* process *)
    ELSE (* ignore *)
    END;
  END VisitCalls;

VAR cu: M3AST_AS.Compilation_Unit;
BEGIN
(* Omitted code to invoke compiler tool to
   compile the unit to tree rooted in 'cu'.
*)
  M3ASTWalk.VisitNodes(cu,
    NEW(StreamClosure, s := StdIO.Out()));
END FindCalls.
```

Object-oriented programming purists may consider the use of `TYPECASE` bad style. An alternative would be to extend the AST with a tool-specific view that added a method to each syntactic node. The only non-trivial implementation of this method, which would be invoked in `VisitCalls`, would be for the `Call` node.

AST Display

All node types in the tree have display methods; for the syntactic nodes the default implementation acts as a pretty printer. Any subtree of an AST can be displayed on a given output stream with an initial indentation. This package is typically used by program transformation tools to recreate a source file from a modified AST.

The Context Manager

Although the focus of attention is typically a single AST, all tools necessarily operate in an environment of multiple ASTs owing to the extensive use of separate compilation in Modula-3. Managing this extra complexity is important to simplify new tool development. This is handled by the notion of a *context*, which is a simple mechanism for collecting together related compilation units. A tool is always provided with a current context so that when a reference to a (named) unit is encountered, for example as the result of an `IMPORT` statement, a search for the unit is first made in the current context. The context manager also provides facilities to iterate all the members of a context. In the current system there is no structure on a context, in particular any subsystem structure is not recorded.

User Interface Management

A tool always contains a user interface component which gathers user input and perhaps generates output. The current state of user interface development is such that, although a window-based interface is often optimal, it is also necessary to provide one based on command lines. It is desirable to hide this distinction from the main body of a tool, and we achieve this by reducing the basic user interface to setting and retrieving typed name-value bindings. We also use extensible stream classes to hide input/output differences. We have implementations of this framework for command lines and for the X window system [Sch88].

5.4 Tools

Any number of tools can be put together into a single program under the control of a master tool that coordinates their activities. Each tool registers itself and its argument requirements with a central manager as part of its initialisation code. Our current toolset includes a compiler front-end, a code-generator, a dependency analyser, a pickle generator and a pre-linker, which can be put together in several combinations. Naturally some tools, such as the compiler front-end, are usually obligatory.

The Compiler Front-End

This is the most complex tool in the environment and, since it is used by almost all the other tools, it is the most important to package cleanly. The tool is structured into three main components, a parser, a semantic analyser and a control module. The parser is written in Modula-3 and uses recursive descent. The parser can parse fragments of Modula-3, for example expressions, which is useful for interpretation and interactive debugging. The semantic analyser is designed as a many-pass system, whereby each pass (tree-walk) typically computes a single attribute. This makes for a very maintainable system but has performance problems. Fortunately we can alleviate this since many attributes can be computed independently; so many tree walks can be collapsed into one simply by changing the driving code to achieve the restructuring. In a multi-threaded environment, many of these passes could be executed in parallel. The parser and semantic analyser are designed to handle a single unit at a time. When an import of another unit is found a callback is made to the driving module, which first tries to find the unit in the context that it was given initially. If that fails it tries to unpickle an AST form of the unit and if none is found recursively compiles the unit from source.

The C-Code Generator

The principal goal for this code-generator was portability, in order to make Modula-3 available on as wide a range of machines as possible. Although we wanted the code to run acceptably fast, if only because the system itself is written in Modula-3, performance was secondary. Although others have reported favourably on using C as an intermediate code [Atk88], we do not share this opinion and would prefer to generate an intermediate representation with more flexibility but without losing portability. Certain aspects of Modula-3, particularly opaque object types and exception handling, are difficult to map efficiently into C. Debugging in C terms is unpleasant, and the mechanisms needed to map information that is only known to the C compiler back into Modula-3 terms are onerous.

The Pre-Linker

The pre-linker operates on the complete set of ASTs that make up a program. It is responsible for generating global information, such as the module initialisation order, and checking the consistent compilation rules. It usually operates as a stand-alone tool, but can be integrated with the compiler. The limiting factor is the size of the ASTs for an entire program; in order to limit the amount of virtual memory and persistent storage needed, the stand alone version operates with *pruned* ASTs which have unimportant information removed.

Pickle Stub Generator

This is the tool that generates Modula-3 code to pickle data structures in a type-safe manner. It is rather like a code generator and operates directly from the ASTs generated by the compiler front-end. In its current form the tool generates type-correct Modula-3 source, which is then compiled by the regular compiler. There are some situations where it would be more convenient (and more efficient) to generate an AST directly, relaxing some type rules, and then feed this directly to the code-generator.

Compile-Servers

One of the problems that faces the implementor of a language with interfaces, like Modula-3, is how to handle the compilation of interfaces. It seems wasteful to continually recompile interfaces as part of compiling a module, which leads designers towards compiled interfaces. Unfortunately, choosing this route places restrictions on the compilation order, since an interface must be compiled before any of the units that import it. This partial ordering can reduce the opportunity for the parallel compilation of a system. We have come to believe that the best solution to this dilemma is to view compiled interfaces exclusively as a performance optimisation. Accordingly, as noted above, our compiler can handle either compiled interfaces or compile them as necessary. So, for example, one might compile the interfaces for the standard library since this is a very stable system, but not those of a system under development. An alternative to pre-compiled interfaces is afforded by a tool that we call a *compile-server*. As the name suggests this is a tool which runs for a long time and responds to compilation requests from a client. In practice it is restricted to handling requests for compilations within a single subsystem (directory), in order to provide a consistent compilation context. The server is merely a standard compiler, outfitted with a mechanism for gathering its arguments and sending its error messages on an IPC channel. The client is a very simple program which redirects its arguments to the channel and listens for replies which it displays on the standard output. A user invokes the client instead of the standard compiler but is otherwise unaware of the existence of the server, other than that compilations execute much faster. In particular it is possible to use this tool in combination with *make* [Fel79] and *emacs* [Sta87] to provide a fast turnround compilation environment.

Dependency Analyser

This tool contains two components, one of which builds a unit dependency graph and another which generates a *Makefile*. Unlike its predecessors [Jor89] this tool makes extensive use of other tools and components. It uses the compiler front-end tool to generate a set of ASTs for the system under consideration and then applies components of the semantic analyser and pre-linker to compute needed attributes, such as the depends-on attribute between modules. The tool can also operate incrementally, monitoring changes to modules in the file system, and regenerating the dependency graph. Since the compiler is integrated with this tool, it is also possible to activate the compiler back-end on changed modules, thus eliminating the *make* step entirely.

6 Related Work

The use of an attributed AST as the basis for tool integration is present in Gandalf [Not85], Mentor [Don84] and the Cornell Synthesiser Generator [Rep89]. These systems are all distinguished by their focus on syntax-directed editing as the program development style, a paradigm that has not found widespread acceptance. They are also more or less closed systems with special purpose programming languages for programming semantic actions. The successor to Mentor, Centaur [Bor88], is a more open system and provides an extensible AST formalism called the Virtual Tree Processor. Unlike M3AST, it is based on a dynamically typed programming language. Another recent intermediate representation is IRIS [Bak87], a graph formalism in which the type of a node is itself defined as an IRIS structure. Much of the work on extensible ASTs dates back to the PQCC project [Lev78]. The IDL language captures many of the essential ideas, but as noted earlier, IDL suffers in the mapping into an operational language with an inadequate type system. Garlan took a very general approach to views [Gar86], and permitted alternative representations of shared data in different views. The idea of composite tools integrated by complex data structures was proposed as an extension to IDL by Snodgrass [Sno86]. The idea of producing tools from tool fragments is advanced in [Ost86]. The evaluation of the Rational system in [Fei88] suggests that it shares many characteristics with our system, although it is rather more ambitious and requires special hardware and operating system support. Objectworks [Par89] for C++ provides similar facilities but details of its internal structure are unknown. The RPDE_{Pascal} [Har89] environment is also similar, but is implemented in a language with no support for object types.

7 Project History and Status

To produce any compiler for a language like Modula-3 is no small task and our timescales did not permit the building of a completely throw-away prototype. So we designed the prototype to support our final objective and built it in such a way that we could translate it into Modula-3, essentially automatically. Since our language of choice was then Modula-2+ [Rov85], this was a plausible approach as the two languages share many features. The critical design decision was the way in which to represent the AST. There is no clean way to support the extensible AST model in Modula-2+. Furthermore, our Modula-2+ implementation provided no pickling mechanism. Our goal was to design an abstract AST specification such that we could map from Modula-2+ to Modula-3 with the minimum of effort. Fortunately, as noted earlier, the IDL language [Sno89] satisfied these requirements very well. We chose to use the UNC IDL toolkit [Sno89] which translates IDL into C. We altered the back-end of the IDL compiler to generate a Modula-2+ interface to the C representation. We began the construction of the prototype system in June 1988. Based on the extensible AST interface, we began parallel development of a compiler front-end, an AST-based interpreter, a debugger and the C-code generator. By October 1988 we could interpret simple programs. The back-end was actually implemented in C at another site and for a long time we communicated persistent AST instances between the two sites. We subsequently enhanced these tools to handle almost the entire language and developed some significant packages, for example an input/output system based on an extensible set of stream classes. As we had hoped, the bootstrap process was almost completely automated by *emacs* editing macros, to convert from Modula-2+ to Modula-3 and also to convert from the IDL form of the AST to the Modula-3 `OBJECT` form. At the time of writing, the system as described is complete and has been distributed to a number of external sites. It is capable of regenerating itself and runs on several different machine architectures. It comprises about 75,000 lines of Modula-3 and 15,000 of C (the C-code generator).

8 Future Work

Now that the basic framework is in place there are many opportunities for new and improved tools. The basic toolset needs symbolic debugging at the Modula-3 level, but this is currently hampered by using C as the target code. Replacing the C-code generator with one targeted to an architecture neutral framework [Dav89], [Pee89], should ultimately provide the same level of portability without the problems posed by C. The set of ASTs for a subsystem or whole program supports browsing to an

arbitrary level of detail, but needs to be coupled with a system modeling capability to control the programming in the large aspects. We believe that we have adequate levels of abstraction in the existing environment so that this could be added with minimal disruption. Whole program optimisation, which acts to undo the levels of abstraction which a language like Modula-3 provides, is another area that can exploit the AST framework. In the longer term two areas seem important to pursue. The first is the replacement of the pickling approach to persistent ASTs with a more incremental approach based on a persistent object store, perhaps an object-oriented database (OODB) with its own type system. There are interesting issues concerned with the performance of a practical programming environment built in such a way and in handling the interaction between a fairly rich type system like Modula-3 and that of an OODB. Second, it seems important to explore how to exploit more formal techniques, for example, attribute grammars, in an open systems framework.

9 Conclusions

We set out to build a practical, extensible program development environment for Modula-3 and we succeeded in reaching that goal. The AST views provide an approachable and well-structured specification and act as a powerful tool integration mechanism. A new tool can be added quickly and easily and very little 'boiler-plate' code is needed in order to exploit other tools and components. The continual striving to produce reusable components paid dividends. Finally, our experience with Modula-3 has been very positive. The structure and quality of the final system owes much to the combination of power and simplicity that Modula-3 provides.

Acknowledgments

David Chase wrote the C code-generator and associated run-time support. Steve Glassman wrote the interpreter and the pickle stub-generator. Trevor Morris wrote much of the compiler front-end and the run-time library. Marion Sturtevant wrote the debugger and the dependency analyser. While the author was unexpectedly between jobs, Sun Microsystems kindly provided facilities for the preparation of the paper.

References

[Aho86] Compilers: Principles, Techniques and Tools. Aho A.V., R. Sethi and J.D. Ullman, Addison-Wesley, Reading, Mass, 1986.

[Atk88] Experiences Creating a Portable Cedar, Russ Atkinson, Alan Demers, Carl Hauser, Christian Jacobi, Peter Kessler and Mark Weiser, Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation.

[Bak87] IRIS: An Internal form for Use in Integrated Environments, D.A. Baker, D.A. Fisher and J.C. Shultis, Technical Report, Incremental Systems Corporation, 1987.

[Bir87] A Simple and Efficient Implementation for Small Databases, A..D. Birrel et al., Proceedings of the Eleventh ACM Symposium on Operating Systems Principles, August 1987.

[Bor88] CENTAUR: the system, P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, V. Pascual, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston Mass, 1988.

[Car89] The Modula-3 Report (Revised), Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. DEC Systems Research Center and Olivetti Research California, November 1989.

[Cra89] A Study of Pickling Emphasizing C++, Daniel H. Craft, Olivetti Software Technology Laboratory Technical Report STL-89-2, September 1989.

[Dav89] A Proposal to the Open Software Foundation for an Architecture-Neutral Distribution Format, J.W. Davidson and T.M. Sigmon, University of Virginia, 1989.

[DOD83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A Edition, United States Department of Defense, Washington DC, 1983.

[Don84] Program Environments based on Structure Editors: the MENTOR experience, in Interactive Programming Environments, D.R. Barstow, H.E. Shrobe and E. Sandewall (Eds), McGraw-Hill, 1984.

[Fel79] Make - A Program for Maintaining Computer Programs, Software, Practice and Experience, Vol. 9, 4, April 1979.

[Fei88] Evaluation of the Rational Environment, P. Feiler, S. Dart, G. Downey, CMU/SEI Technical Report 88-TR-15, July 1988.

[Gar86] Views for Tools in Integrated Environments, David Garlan, in Advanced Programming Environments, LNCS 244, Springer-Verlag, 1986.

[Goo83] DIANA, An Intermediate Language for ADA, Lecture Notes in Computer Science, 161, Springer Verlag, 1983.

[Gut85] Larch in Five Easy Pieces, J.V. Guttag, J.J. Horning and J.M. Wing, Research Report 5, DEC Systems Research Center, Palo Alto, CA, 1985.

[Har89] Good News, Bad News: Experience Building a Software Development Environment Using the Object-Oriented Paradigm, W. Harrison, P. Sweeney, J. Shilling, OOPSLA Conference Proceedings, October 1989.

[Her82] A Value Transmission Method for Abstract Data Types, M. Herlihy and B. Liskov, ACM Trans. on Programming Languages and Systems, October 1982.

[Jor88] A Programming Environment for Modula-2, Mick Jordan and Peter Robinson, Software Engineering Journal, 3(4), July 1988, pp119-126.

[Jor89] Experiences in Configuration Management for Modula-2, Mick Jordan, Proceedings of the 2nd International Workshop on Configuration Management, Princeton, New Jersey. ACM SIGSOFT Software Enginerering Notes, 17, 7, Nov 89.

[Jor90] A Space Efficient Representation for Sequences in Abstract Syntax Trees, Mick Jordan, Unpublished Technical Report, Feb 1990.

[Ker78] The C Programming Language, B. Kernigan and D. Ritchie, Prentice Hall, 1978.

[Lev78] An Overview of the PQCC Project, B.W. Leverett, R.G.G. Cattell, S.O. Hobbs, J,N. Newcomer, A.H.Reiner, B.R. Schatz and W.A. Wulf, Carnegie Mellon University, Pittsburgh, PA, 1978.

[Mey88] Object-Oriented Software Construction, B. Meyer, Prentice Hall, 1988.

[Neu88] C. M. Neuwirth and A. Ogura, Programmers Guide to the Andrew Toolkit, CMU Information Technology Center, January 1988.

[Not85] Special Issue on the GANDALF Project, D. Notkin, R.J. Ellison, B.J. Staudt, G.E.Kaiser, E. Kant, N. Habermann, V. Ambriola and C. Montagero, Journal of Systems and Software, 5, 2, May 1985.

[Ost86] A Process-Object Centered View of Software Environment Architecture, L. Osterweil, in Advanced Programming Environments, LNCS 244, Springer-Verlag, 1986.

[Par89] Objectworks, for C++, ParcPlace Systems, Mountain View, CA 94043, 1989.

[Pee89] Ten15 Distribution Format, N. Peeling, Royal Signals and Radar Research Establishment, Malvern, Worcs, England, 1989.

[Rep89] The Synthesizer Generator, T.W Reps and T. Teitelbaum, Springer-Verlag, New York, 1989.

[Sch88] X Window System: C Library and Protocol Reference, R. Scheifler, J. Gettys, R. Newman, Digital Press, Bedford, MA, 1988.

[Sno86] Supporting Flexible and Efficient Tool Integration, R. SnodGrass and Karen Shannon, in Advanced Programming Environments, LNCS 244, Springer-Verlag, 1986.

[Sno89] The Interface Description Language: Definition and Use, Richard Snodgrass, Computer Science Press, NY, 1989.

[Sta87] GNU Emacs Manual, R. Stallman, Free Software Foundation, March 1987.

[Str86] Stroustrup, B. The C++ Programming Language. Addison-Wesley, Reading, Mass.

[Tes85] Object Pascal Report, L. Tesler, Structured Language World, 9(3), 1985.

[Rov85] On Extending Modula-2 for Building Large, Integrated Systems, Paul Rovner, Roy Levin, John Wick. DEC Systems Research Center, Palo Alto, January 1985.

[Wil88] PGRAPHITE: An Experiment in Persistent Typed Object Management, J.C. Wileden, A.L. Wolf, C.D. Fisher, P.L Tarr, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, 1988.

[Wir83] Programming in Modula-2, 3rd Edition, Texts and Monographs in Computer Science, New York, Springer Verlag, 1983.

[Zor85] Experiences with Ada Code Generation. B.G. Zorn, Technical Report UCB/CSD 85/249, University of California, Berkeley, June 1985.