ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4145

Title: The SMITE Modular Compilation System

Author: C L Harrold

Date: March 1988

SUMMARY

This paper gives an outline of the ideas underlying the Modular Compilation System to be used in the SMITE secure computer. This system prevents users loosing track of which generation of source text gave rise to which object code, by binding both together into an abstract data object, called a module. Operations give access to the latest versions and require both to be updated together. Linking modules together to form an executable object is performed automatically, so it becomes impossible to forget to relink after updating a module. Configuration control and mixed language working can also be provided.

1. Introduction

The SMITE Modular Compilation System provides an environment for software development and maintenance. At the lowest level it binds together source text and object code into modules. The source text and object code can only be updated together, so it is not possible to loose track of which generation of source text gave rise to which object code.

The concept of separately 'linking' a program together to make a runnable object has no meaning in a capability based architecture [5]. Whenever a program is executed the most up to date versions of all the necessary modules are automatically used. Consequently, it becomes impossible to forget to relink a program after modules have been altered, and there is no confusion as to which source text gave rise to the executable program.

Modules which are a binding of more than just the source text and object code may be defined, and used to provide change control histories and facilitate version management of complete sub-systems. By defining interface modules to convert between different languages specification formats, mixed language working can also be provided.

SMITE relies on strong typing using abstract types to achieve the high levels of assurance necessary for both developing and using secure systems [4]. An abstract type is one which is defined solely in terms of the operations that can be applied to it. The underlying representation of the object can never be discovered or accessed other than via their operations. Refer to [6] for a more detailed explanation of the protection mechanisms of SMITE.

The modular compilation system used in SMITE is largely based on that of the FLEX capability computer [1], which has also been used in the Ten15 software engineering environment [3].

2. Modules

This section describes the basic underlying structure of modules in SMITE. Modules as described here will not, in general, be accessible to the users, but will form the basis of the language modules that the users can create and use (refer to section 6, language modules). However, it may be the case that certain security critical code may wish to use these basic modules and avoid the complications of language modules.

2.1 Structure of a Basic Module

In very simple terms, a module is a procedure which may create and initialise some objects and deliver a capability for a selection of these. This result consists of the objects explicitly 'kept' or exported by the module and is known as the 'keeps'. In order to prevent accidental or malicious change to these keeps, what is in fact delivered is a readonly capability to a data block containing the objects. Modules may 'use' or import the results of other modules in the construction of their results.

Modules are essentially variables that bind the simple procedure outlined above, the object code, with the source text and some notion of which user 'owns' the module. Modules are persistent, and can be held in the backing store [8]. At the lowest level a module is implemented by an abstract data type, called a **BasicModule**. The operations defining a BasicModule allow access to the source text and object code, and provide a means to update both simultaneously.

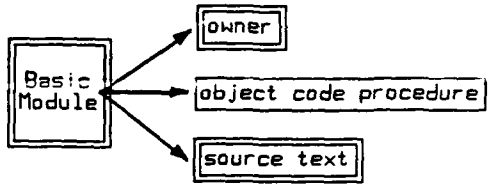

Figure 1: Abstract Data Type - Basic Module

Note that objects with two enclosing boxes represent abstract objects, and a single box represents primitive objects.

The owner of a Module is a **User**. This is an abstract data type to represent the users of the system. The owner of a module is the only user entitled to update it. No operations to give access to the owner of a module will be defined, as this information could be used maliciously to alter the module. However, an operation to check whether a particular user is the owner of a module is defined.

The source text of a module is contained in an object of type **Edfile**. This is an abstract data type for an unalterable editable file. Edfiles are structured objects and can contain capabilities for objects, such as modules. Once created, edfiles cannot be altered. This means that the source text of a module cannot be altered indirectly.

The object code of a module is the procedure which constructs and delivers the keeps. In order that the keeps for each module are only generated once, and that no procedure can masquerade as the object code and deliver false keeps, the object code procedure must be given a **Loader** as a parameter. The object code procedure will use the loader that it was passed to load the object code of any modules that it uses, gain access to their keeps, and then construct its own result. A Loader is a further abstract type, and can only be created in the prescribed manner (refer to section 3, Loaders). Note that the loaded object code will always be the most recent assignment to the module.

2.2 Operations on Basic Modules

new: (Edfile × (Loader→PtrX) × User) → BasicModule X
New modules are created from an editable file containing the source text, the corresponding object code procedure and the owner information. All modules created by 'new' are unique, even if the source text, object code and owner are identical to another module.

get_text: BasicModule X → Edfile
The operation 'get_text' returns the editable file which contains the source text of the module.

get_code: BasicModule X → (Loader→PtrX)
The operation 'get_code', when applied to a module, delivers the object code procedure.

amend: (BasicModule X × User × Edfile × (Loader→PtrX)) → ()
Modules are variable objects so that they can be altered. The operation 'amend' takes the module to be updated, the identity of the user performing the operation, the new source text and object code procedure and performs the assignment. This operation will fail unless the user is the owner of the module, and the 'types' (basically the type of module's keeps) of the new and old object code procedures are the same. After the assignment, the get_text and get_code procedures will give the new versions of the source text and object code.

check_owner: User × BasicModule X → Bool
This operation checks whether a particular user is the owner of the supplied module.

is_module?: Capability → Bool
This operation checks whether the supplied parameter is a capability for an object of type 'basic module'. Note that this operation will only be necessary if the general type checking is not sufficiently trustworthy.

3. Loaders

This section describes the abstract type Loader, that is used for loading modules and automatically linking program.

3.1 Structure of a Loader

The only operation that can be applied to a Loader is to open it. This results in a procedure for loading basic modules. This 'loader_procedure' takes a basic module as a parameter and delivers its keeps. The usual action of this procedure is to call the object code procedure from the module with the unopened Loader, and deliver the resulting module keeps. In order to do this, one of the non-locals of the loader_procedure will be the original Loader, that is the loader_procedure knows which Loader it was created from.

So that modules are only loaded once, the loader_procedure maintains a list of those modules it has already loaded and their associated keeps. Then, whenever the loader_procedure is called with a module that is in the list, the keeps can be delivered.

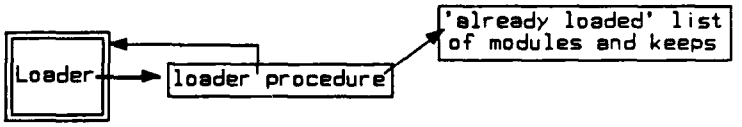

Figure 2: Abstract Data Type - Loader

In order to detect, and fail, recursive usage of modules, they are added to the 'already loaded' list before the object code procedure is called. The keeps delivered by the object code procedure is then used to update the list and delivered as the result of the load. Consequently, if an attempt is made to load a module which is in the list, but with no keeps, it can be failed.

It is often required that a loader is not created with an empty 'already loaded' list, but that it contains common modules, such as the editor. To this end the type, **Loadlist** is defined. An operation is defined to create an empty list, and it is also possible to recover the list from a Loader. Loadlists must be protected from accidental or malicious change, in particular to prevent loaders being created which supply false keeps for important modules, therefore Loadlist must be an abstract type.

In general, users will not explicitly create or use Loaders as this will be done automatically when a user requests execution of a program, refer to section 4. Program.

3.2 Loader Operations

new_list: () → Loadlist
This operation creates an empty loadlist.

open: Loader → (BasicModuleX→PtrX)
This operation, described above, opens a loader allowing the loader_procedure to be called to load a module.

new_loader: Loadlist → (Loader × (()→Loadlist))
This procedure will create a unique new loader with the supplied parameter making up the 'already loaded' list of modules and keeps. A procedure to recover the list is also delivered.

4. Program

This section describes the abstract data type, **Program**, that is used to 'hide' basic modules, and create runnable objects.

4.1 Structure of Program

Once a Program has been created the only possible operation that can be applied to it is 'load', which results in the keeps of the hidden module. The module itself can never be recovered from the Program, not even by the originator.

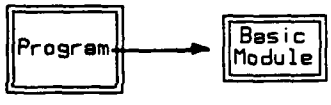

Figure 3: Abstract Data Type - Program

Although the hidden module could keep any type of object, a Program is usually created from a module that keeps a procedure. The loading operation will then result in a callable procedure. Once a program has been loaded the resulting object can be called with suitable parameters as often as required, but will not be initialised (ie loaded) again.

'Running' a Program comprises loading the hidden module and then applying the resulting procedure to suitable parameters. Every time a program is run afresh, the module will be loaded and initialised, and consequently the most recent assignments to modules will always be used. The loader used could be created from a new Loadlist, but it is generally expected that the 'current' Loadlist is used. This is the 'already loaded' list from the loader that was used to load the command shell and consequently already contains many common modules.

This wrapping up of modules into objects that can only be loaded is mainly for protection. However, it could also be useful that the 'programs to be run' and the 'modules' they were created from are recognisably distinct. These program objects can be freely supplied to a software developer or other users of the system, who may then use the procedures they keep, that is 'run the program', without having any access to the underlying module or source text.

4.2 Program Operations

make_program: BasicModule X → Program X
This operation hides a Basic Module by wrapping it up into an abstract data type.

load_program: Program X → Ptr X
This operation will load the program object by calling the object code procedure from the hidden module with a loader as a parameter.

5. Implementation

This section gives an outline of how the Basic Modules, Loaders and Program described in the previous sections will be implemented, and illustrates the mechanisms using a simple example.

5.1 Implementing Abstract Types

Abstract data types can be efficiently implemented by the SMITE computer architecture using keyed blocks and procedures [6&7].

Keyed blocks contain data and a key and can be locked by a single instruction. The data stored in a locked keyed block can only be accessed by unlocking the block with the correct key. This unlocking is also implemented by a single instruction.

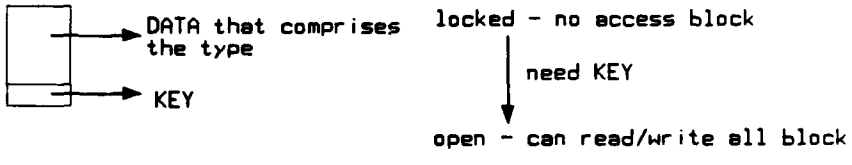

Figure 4: A Keyed Block

Procedures in SMITE are first class data objects [2] which can only be called. They bind together code and non-local data, which is only accessible to the running procedure. Thus procedures offer information hiding. Procedures can be stored on the backing store and will be automatically copied into main store whenever they are called.

Abstract types can, then, be implemented by hiding the data that comprises the type in a keyed block, and supplying the operations as procedures. These procedures will either take a locked keyed block as a parameter (along with other data), or create a new one. The non-local data of these procedures will contain the key for the block (the type manager key). The code of the procedures must be trusted not to disclose the key under any circumstances, nor to give access to unlocked keyed blocks.

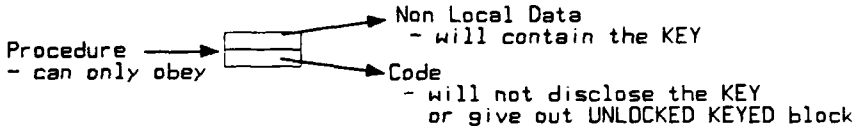

Figure 5: Abstract Type Operations as procedures

Creating a type manager for an abstract type will therefore involve generating a unique new key (this will be a capability as these are guaranteed to be unforgable and unique), and creating the procedures to perform the required operations. The key will be bound into the non-local data of these procedures. Once these procedures have been certified not to disclose any information, they may be safely installed in the system.

5.2 Implementing Basic Modules

Basic Modules will be implemented as keyed blocks and procedures as described above. Along with the key, the block will contain capabilities to keyed blocks representing the owner of the module and the source text edfile, and the object code procedure. Procedures taking such a block as a parameter will implement the operations on the type. These will know the key, and can open the block and access the data as required.

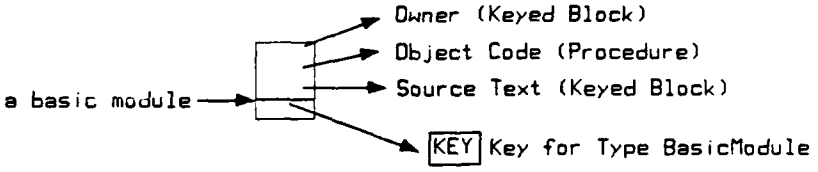

Figure 6: Keyed Block for Basic Modules

new: (Edfile × (Loader→PtrX) × User) → BasicModule X
Creating new basic modules will involve generating a new four word keyed block and then storing the key, source text, object code and owner into it. The pointer to this block will then be locked and delivered.

get_text, get_code: BasicModule X → Edfile, BasicModule X → (Loader→PtrX)
These procedures will open the keyed block with the key and can then deliver the appropriate word from the block.

check_owner: User × BasicModule X → Bool
This procedure will open the keyed block and then perform an equality operation between the user in question and the owner stored in the block.

is_module?: KeyedBlock → Bool
This procedure attempts to open the keyed block parameter with the key for the type 'basic module'. If this fails then the keyed block does not represent a basic module.

amend: (BasicModule X × User × Edfile × (Loader→PtrX)) → ()
Subject to the check on the owner and type of the object code procedure, this procedure will open the keyed block with the key, and replace the source text and object code with the new versions.

These procedures may wish to check that the objects they have been passed as parameters represent the expected objects. This would be achieved by including other procedures, such as is_edfile? or is_user?, in their non-local environments.

5.3 Implementing Loaders

The type Loadlist can be implemented as a keyed block consisting of a key and a capability for an ordinary data block containing the list of modules and keeps. The operation to create a new loader will know the key so that it can recover the list. The 'already loaded' list in the loader_procedure will not be of type Loadlist, so that the adding and checking operations do not need to be implemented as abstract type operations.

Loaders are implemented in the SMITE Instruction Set by open pointers to closure blocks [7]. The closure will contain the code and non-locals of the loader_procedure. Closure blocks are always generated with a locked pointer and the ability to open them is a single, privileged, instruction. This is only used by new_loader. There is a second instruction to open the loader. This first checks whether the pointer is open (ie represents the abstract type 'Loader'), and will fail if it is not.

Note that both locked and open pointers to closure blocks may be called, and it is therefore important that open_loader is always used to check that the object really does represent a loader.

5.4 Implementing Program

Since procedures offer information hiding, Program can be implemented as a procedure with the basic module in its non-local environment.

The code of the program procedure will generate a loader (either with an empty 'already loaded' list or using the current one) and load the hidden module by calling its object code procedure with the loader as the parameter, delivering the keeps. This implements the load_program operation. The get_code procedure and the procedure to create a loader will also be required in the non-local environment. However, there is no need for this code to have access to any type manager keys. The code of the procedure must, of course, be trusted not to disclose the hidden module under any circumstances.

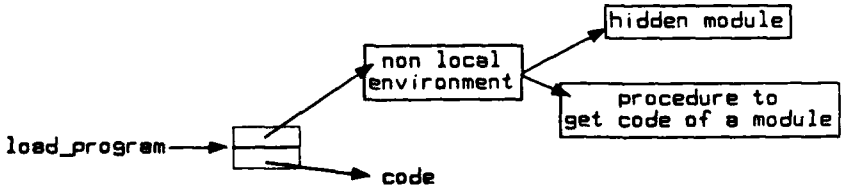

Figure 7: Implementing Program

The operation to make_program will also be a implemented as a procedure. This procedure will form some non-local data with a basic module (and any other necessary procedures), and close it with the load_program code to create the program procedure described above.

5.5 Example

Consider as an example defining a procedure that takes a string and an integer as parameters, calls a procedure that returns the textual representation of the integer, and delivers the combined string as a result.

```
test_module : ((String×Int)→String)
```

Source Text:

```
test_module:

capability for module defining STRING and +
capability for module defining intchars

PROC test = ( STRING a, INT b ) STRING:
BEGIN
   a + intchars( b )
END

KEEP test
FINISH
```

Capabilities for the required modules are placed in the use list. These modules will keep the definition of the type STRING, the declaration of the procedure intchars (that turns integers into strings), and the declaration of the operator + between strings.

The object code procedure for this example module would be along the lines of the following procedure.

non-local data:

```
get_code procedure
capability for first used module
capability for second used module
```

code:

```
PROC object_code = ( LOADER loader ) KEEPS:
BEGIN
   IF loader can be opened
   THEN
      {get the object code procedures of the used modules
      and call them with the loader}
      PROC(LOADER)KEEPS one = get_code( first non-local );
      OP (STRING,STRING)STRING + = one( loader );

      PROC(LOADER)KEEPS two = get_code( second non-local );
      PROC(INT)STRING intchars = two( loader );

      {construct the procedure that uses these and deliver it}
      PROC test = ( STRING a, INT b ) STRING:
      BEGIN
         a + intchars( b )
      END
      test
   ELSE
      fail
   FI
END
```

This module can be turned into runnable program by using the make_program operations. This will create a procedure which has the module as a non-local and code to load the module.

```
Program( (String×Int)→String )

  non-local data: get_code procedure
                  test_module : ((String×Int)→String)

  code: create a loader
        PROC(LOADER)KEEPS code = get_code( non-local module )
        code( loader ) {delivers the keeps}
```

6. Language Specific Modules

In high level languages, modules may use objects created by other modules, and provide objects for use by further modules. A module must obviously provide the names and types of these objects to allow type checking to be performed. This information forms the specification of a module. Language specific modules are implemented by combining basic modules with specifications, the particular format depending on the language.

Users of SMITE will wish to create modules in various languages, such as Algol68, Ada and ML. However, language modules cannot be freely mixed. If mixed language working is required, interface modules will be necessary to convert between the different languages' specification formats.

6.1 Compiling Source Text

Modules are created by compiling source text. Compilers in SMITE will be applied to editable files containing the source text written in a particular language and will deliver abstract objects, called **compiledSpecifics**. These compileds are simply a binding of the source text, object code and specification, and are abstract objects in order to prevent users or untrusted software altering them. Compileds can then be either made into a new language specific module or assigned to an existing one. For example the Algol68 compiler will produce CompiledAlgol68 which could then be made into a new Algol68Module.

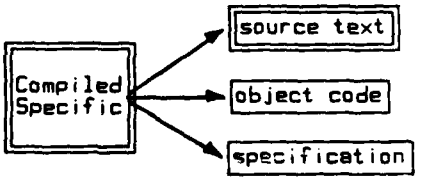

Figure 6: Abstract Data Type - CompiledSpecific

The only operations defined on compileds, apart from creation, will be those to recover the source text, object code and specification. Compileds cannot be altered in any way.

6.2 Structure of Language Modules

Language Specific Modules are further abstract types. They are a binding of a basic module, which contains the source text, object code and owner, with a specification. Language specific modules are persistent variables and can be assigned to by either the procedure 'amend' or 'change_spec', depending upon whether the specification will be altered or not. Only the owner of a module will be permitted to perform assignments.

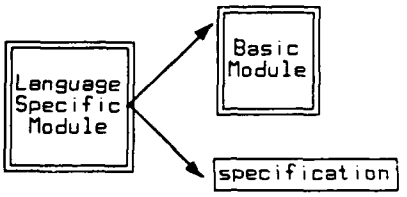

Figure 7: Abstract Data Type - Language Specific Module

6.3 Operations on Language Specific Modules

newSpecific: User × CompiledSpecific → SpecificModule
This operation will create a new basic module from the user parameter and the source text and object code in the compiled, and bind this with the specification from the compiled to create a unique new language module. Note that two modules created from the same compiled by the same user will be different objects.

get_Specificcode: SpecificModule × SpecificSpec → (Loader→PtrX)
This operation will deliver the object code procedure of the supplied language module. The parameters to this procedure are the language module and its expected specification. Whenever the language module has had its specification changed, the expected specification will differ from the specification bound into the language module, and a suitable exception will be raised.

Note that comparison of specifications is performed in a language specific manner, and is not necessarily simply equality. This is important for languages such as ML.

amendSpecific: User × CompiledSpecific × SpecificModule → ()
Subject to the user and specification constraints, this operation will amend the underlying basic module with the new source text and object code from the compiled. All users of the module will immediately see the change, and be supplied with the most up to date versions.

change_specSpecific: User × CompiledSpecific × SpecificModule → ()
Subject to the constraints on the User, this operation will create a new basic module with the source text and object code from the compiled, and use it to replace the existing basic module in the supplied language module. The new specification from the compiled will be used to replace the existing one. This effectively updates all uses of the module.

Any modules that use this 'changed' module will now need recompiling, and cannot be used until this has been performed. This can be detected, and performed automatically by the recompile_me operation described below.

recompile_me: SpecificModule × SpecificSpec → Bool
This operation takes a language specific module and its expected specification as parameters, and returns a boolean to indicate whether the specifications are incompatable. In addition, the recompile_me procedures of all the used modules are called with their expected specifications (that is, the specifications at the compile time of the supplied module). Finally, if any of the used modules expected and actual specifications are incompatible, the appropriate compiler is applied to the source text of the supplied module and the result assigned to it. This will then ensure complete consistency.

```
PROC recompile_me = ( MODULE me, SPEC s ) BOOL:
BEGIN
   BOOL altered := FALSE;
   FORALL used module
   DO
      altered :=
        altered OR recompile_me( used module, expected spec )
   OD
   IF altered
   THEN
      compile my source text and assign it to me
   FI
   s = my spec
END
```

Note that section 7 explains how recompile_me can be implemented.

get_Specifictext: SpecificModule → Edfile
This operation will deliver the source text from the underlying basic module.

get_Specificspec: SpecificModule → SpecificSpec
This operation will deliver the specification from a language specific module.

get_basic: SpecificModule → BasicModule X
This operation will deliver the underlying basic module from a language specific one.

is_Specificmodule?: Capability → Bool
This operation will check whether the supplied parameter represents an object of type SpecificModule.

The following are also important operations involving language specific modules. However, these are not defined as abstract type operations as they are simply a combination of already defined operations.

LoadSpecific: Loader × SpecificModule → Ptr X
Loading a language specific module will involve applying a loader to the object code procedure from the basic module. If this module uses other language modules the same loader will be applied to these.

Note that the operation to get the object code procedure from a language module also checks that the specification has not been changed. Therefore, inconsistent modules will fail to load.

Make_SpecificProgram: User × SpecificModule → Program
It is required that whenever any of the modules used in a program have their specification changed, that the program cannot be executed. The get_specificcode operation first checks the specifications, and therefore inconsistent program will fail to load. However, in order to be able to discover if the specification of the hidden module (that is the module the program was created from) has been altered, Program cannot be created from the underlying basic module of a language specific one, as no specification would be available. It would be possible to define the type Program to hide language specific modules as well as basic ones, but this would not allow new languages to be easily introduced into the system.

Instead, program will be created from a new basic module. The object code procedure of this new module will contain the language specific module and specification in its non-local environment, and its code will perform the necessary specification checks and load the module. This new basic module can then be made into Program in the usual way using the make_program operation, as described in section 4. This mechanism will ensure that Program need not be changed whenever the hidden module is amended, but will fail to load, and consequently cannot be executed, if the specification is changed.

6.4 Extending the Language Specific Modules

In some cases, it may be desirable to hide the text of a language specific module. Simply discarding the source text from the basic module would be insufficient, as whenever the module was updated the source text would be replaced. Instead a mechanism for 'hiding' the aspects of a module is required.

This hiding can be achieved if the operations on a language module actually called procedures to deliver the required objects, rather than simply recovering the objects themselves. For example the get_specifictext operation on a language specific module would call a procedure that would generally supply the source text from the underlying basic module. New language modules could then be created which were simply an 'indirection' to the original module through these procedures (ie contain the module in their non-local environments). Changes to the original module would then be tracked, but the code of the get_text procedure would still refuse to disclose the source text. This indirection to a module is a very powerful mechansim and could be used in various ways.

It may be advantageous to keep a history of changes to a module. This could record when changes were made and why they were necessary, as part of the overall software development. This could be achieved by extending a language module to consist of a 'history' as well as the basic module and specification. The assignment operations would be extended to add to this history and extra operations would be provided to act upon it as required. The following diagram indicates a possible format.

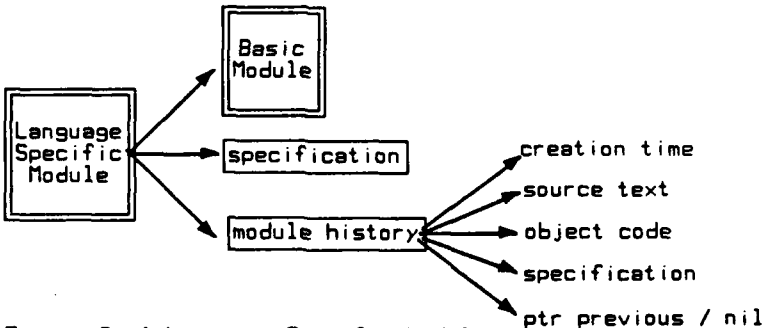

Figure 8: A Language Specific Module with History

It would also probably be desirable to define an operation on language specific modules to provide the compiler name. This would then be used when displaying modules to indicate the type to the user.

7. Implementing Language Modules

As for the basic modules, language specific modules will be implemented by keyed blocks and procedures. However, in order to facilitate 'hiding' aspects of a module, the keyed block will not contain the underlying data objects, but procedures to supply them.

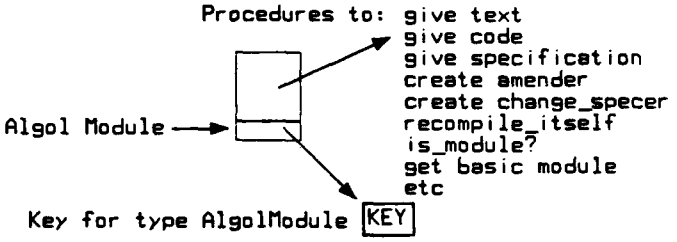

Figure 9: Algol Module

The recompile_me procedure needs to know the used modules and their specifications. The modules used could be discovered by examining the source text. However, the compile time specifications would not be available. All the necessary information is in fact contained in the non-local environment of the object code procedure, and it could be arranged that a capability to this structure is also supplied by the compiler and kept as part of the module data.

The procedures implementing the operations on the type will need to have the underlying basic module, specification and used modules structure as well in their non-local environments. These are bound together into an ordinary data block, which the procedures are trusted not to disclose. Some of the operations will also require access to the procedures that operate on basic modules, but they will not need to know any keys.

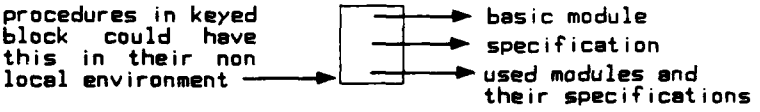

Figure 10: Data for Language Modules

Note that in mixed language working, the recompile_me field of the interface module will need to be be an indirection to the required procedure, in the same way as hiding text is an indirection.

8. Module Management and Configuration Control

Systems generally need to have some form of Module Management and Configuration Control built into them. The ideas described in the previous sections ensure that the most up to date source text and object code are always used, and that whenever a module has its specification changed, all the necessary modules can be automatically recompiled. However, it is often the case that various versions of a system would be required, for example 'experimental version' and 'working version'. This could be achieved by creating a further type of module which is an indirection to the various versions of the required language module.

A version module could be as below. However, the exact structure of a version module and the particular operations defined on them on then, would depend upon the particular requirements of the system.

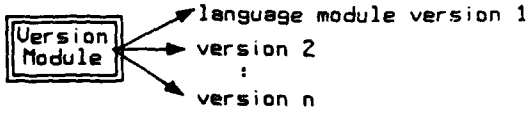

Figure 11: a version module

The SMITE Instruction set will allow dynamic process contexts to be set. There are instructions to set the process context and this can be done when creating program objects. Whenever a version module is to be loaded, the loader_procedure will check the process context for the version number and simply load the appropriate one.

Users will wish to create modules in various languages and will require that a record is kept of all the modules they have created, even those not currently in use. A user will then wish to be able to find all these modules and to delete any unwanted ones. This could be done by having 'Module Sets' associated with the users.

A Module Set could be a linked list of language specific modules. These will be in various languages, and so each user might have a list for each language used. Whenever new modules are created an entry would be added to the list. Operations could then be defined to show all the modules in a particular set, discover which modules are not used and to delete a module from the set if it is no longer required.

In some systems, it may be required to journal the accesses to a module or the amendments that a user has performed. A journal structure could be added to the language module, or provision could be made in the abstract data types that describe the users of SMITE as appropriate.

9. Conclusions

The SMITE Modular Compilation system will provide an environment for the development and maintenance of software for secure systems. A high assurance implementation is possible because abstract types are used to divide the code into many small independent parts. The protection offered by this approach is flexible and is efficiently implemented on the SMITE computer architecture.

References

[1] FLEX: A Working Computer with an Architecture Based on Procedure Values.
J M Foster, I F Currie, P W Edwards
RSRE Memo 3500, July 1982

[2] In Praise of Procedures
I F Currie
Memo 3499, July 1982

[3] Ten15: An Overview
P W Core and J M Foster
RSRE Memo 3977, September 1986

[4] The Development Environment for Secure Software
C T Sennett
RSRE Report 87015, November 1987

[5] The SMITE Computer Architecture
S R Wiseman & H S Field-Richards
Memo 4126, January 1988

[6] Protection and Security Mechanisms in the SMITE Capability Computer
S R Wiseman
RSRE Memo 4117, January 1988

[7] The SMITE Instruction Set Specification
C L Harrold and S R Wiseman
RSRE Memo in Preparation, 1988

[8] The SMITE Object Oriented Backing Store
S R Wiseman
RSRE Memo in Preparation, 1988

Acknowledgments

Thanks to Simon Wiseman, Philip Core and Peter Bottomley for their comments and suggestions.

DOCUMENT CONTROL SHEET

Overall security classification of sheet Unclassified

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Memorandum 4145		Unclassified

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
	Royal Signals and Radar Establishment St Andrews Road, Malvern, Worcestershire WR14 3PS

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

The SMITE Modular Compilation System

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
Harrold C L			1988.03	15

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

This paper gives an outline of the ideas underlying the Modular Compilation System to be used in the SMITE secure computer. This system prevents users loosing track of which generation of source text gave rise to which object code, by binding both together into an abstract data object, called a module. Operations give access to the latest versions and require both to be updated together. Linking modules together to form an executable object is performed automatically, so it becomes impossible to forget to relink after updating a module. Configuration control and mixed language working can also be provided.