*J.King Saud Univ.*, Vol.9, *Computer & Information Sci.*, pp. 21-40 (A.H 1417/1997)

# A Logic-based Framework for Software Construction

**Abid Jazaa**
*Computer Science Department, Yarmouk University, Irbid, Jordan*

(Received, 06 Sept. 1993; accepted for publication, 09 Oct. 1995)

**Abstract.** This paper describes a framework for assisting the automatic construction of programs from separately designed and compiled modules. The framework has facilities to automatically extract many important items of information from source codes. It has also facilities to deduce new items from old ones. The framework also supports multiple versions, static, and dynamic selections of appropriate components. It supports graphical displays for system's components and their relationships.

## 1. Introduction

A typical software system includes thousands of components spread over a large number of modules. The relationships among these modules are difficult to determine and hard to visualize. In addition, any bug-fixing, adaptation, and enhancement of a module may introduce unexpected interactions between diverse parts of the software system. This problem becomes more complicated when multiple variations are supported particularly for large software applications [1, 2]. The construction of a software system, out of these multiple module variations, becomes tedious and time consuming without the support of automatic aids.

We propose a method to construct programs from pre-tested modules. It is developed within a framework for software automation and version control. This new method which is unlike previous facilities [1] provides aids for selecting the required version of a component and for tracing the dependencies between components. This allows the systematic building of systems and/or sub-systems. This method also provides rules that embody human decision making activities.

The paper is structured as follows: a literature review is conducted in section 2. The basic structure of the proposed framework is discussed and described in section 3,. Dependencies, relationships, and version selection are also outlined in section 3. Compilation strategies are discussed in section 4. Automatic Building of software systems is discussed in section 5. The module invocation algorithm is described in section 6. Section 7 represents working examples. Conclusions are drawn and future works are identified in section 8.

## 2. Background

A software construction tool (or assistant) employs methods that can be used to construct a software system out of many separately compiled modules. DeRemer and Kron [3] were the first to introduce the basic ideas and concepts of software construction tools.

One of the first tools used widely for building software systems was the UNIX tool Make. Make is based on recording the dependencies among program parts and, by examining the times at which each module was created (or modified), it can decide what parts need to be re-generated and what parts remain valid. The fundamental source of information for Make is explicit rules of dependencies and actions (it also provides a simple macro expansion facility and a set of built-in rules for commonly related files). This information is provided by the user in a description file known as the makefile. However, Make has a number of drawbacks that limit its applicability particularly for software construction. These limitations include:

- it relies on rules supplied by the user. It is generated manually and therefore it is impractical for large software systems that are made up of thousands of entities participating in many different types of relationships and held in a knowledge base,
- makefile is not immediately accessible to a general purpose query language to analyze the amount of work that has to be done if a particular component is changed. It is also based on the 'file-name' model of naming,
- it is binary oriented - the user must describe the system in terms of the object modules, rather than in terms of source modules,
- it was developed to support automatic building only, and is therefore limited in application to certain parts of the software life cycle[4],
- it depends on the underlying operating system to handle item identities, and this on its own is a serious drawback of the tool[4],
- it has no facilities to maintain integrity of the system configurations[5].

A number of make-like methods (or extensions) have been proposed but they solve a subset of the problems in ad hoc ways and as described by Baalbergen et al. [6] "place the burden of writing complex description files on the programer's shoulders".

Most existing MILs have failed to address version representations, for full details see the surveys published in ([7], [8]). Stronger interconnection models are still required and this is an area for further research as Tichy [9] claims: "Most MILs suffer from not treating interfaces as first-class software objects. Thus, it is difficult to represent versions of interfaces. This is a serious limitation, even though versions of interfaces do not arise as frequently as versions of implemented programs".

A number of practical software systems claim that they incorporate facilities for constructing software systems (see for example; Changemen, CMS, Lifespan or Perspective which are outlined in the START Guide 1987 [4]). The above practical SCM systems do not make clear what is meant by a building process. It is similar to composite versions adopted by the PCTE (+) interfaces. For example, Changeman (Changeman, Technical Overview, SD EuropeLtd, 1989) explains the build process in the following way. "This process groups released items into sets which can be referred to by a single item name. Whilst only released items can be built, single items may be included in several builds. The build process will accept items which themselves are configurations, thus building a hierarchy".

There are a number of interesting software disciplines that may be related to our work directly or indirectly such as Object-Oriented techniques, Artificial Neural Networks, and software environment tools. One of the famous environments available is the Computer Assisted Software Engineering (CASE). CASE tools are essentially a product of the mid-1980s, the degree of sophistication ranging from being a drawing tool to being a tool that balances and checks models as well as generating source code[10]. CASE has fallen so short of its promise[1]. The problem of CASE tools seems to be on both sides of the fence as Loy, an experienced user of CASE tools, reports [11]: "The user's expectations of CASE have been unrealistic, and the products themselves have not delivered what the vendors said they would".

## 3. The Basic Structure of the Proposed Framework

The proposed system is based on a directed acyclic graph where nodes represent modules and arcs represent interfaces. The proposed system is a collection of highly interacting modules. We assume that a typical module has two parts; an interface and an

implementation where each interface or implementation may have unlimited number of releases. We assume, further, that the interface has two properties [12] :

separable: the modules in a system can be designed and verified separately,
composable: the proof obligations are to show that each module in a system satisfies the module's interfaces.

Modules and their releases can be represented as *n-tuple*. Modules A1, ..., An, n > 1 , the Cartesian product of A1, ..., An, denoted by A1 X A2 X ... X An, is defined:

$$A1X\ A2\ X\ ...\ X\ An = \{ (x_1, ..., x_n) \mid x_i \text{ is in } Ai \quad 1<= i <= n\}$$

Any system can be represented as a digraph of the form :

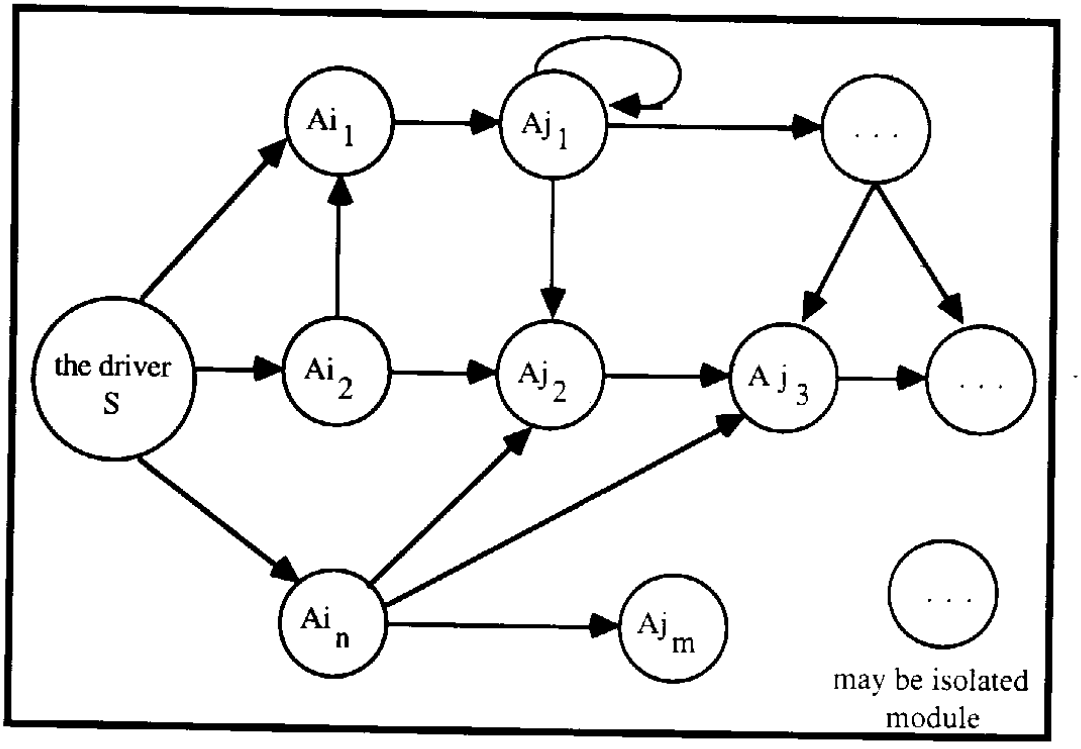


**Fig. 1. System diagram of the form.**

Given a system, S, define a digraph G of an ordered pair G = (V, E) of vertices (modules) V and directed edges ( invocation) E that belongs to the Cartesian product VXV, where V and E are finite sets. Given two modules $Ai_k$ and $Aj_r$ in a digraph G = (V, E), then Aik is a predecessor of Ajr and Ajr is successor of Aik if there is an edge

( $Ai_k$ , $Aj_r$) in E i.e. if and only if module $Ai_k$ contains an invocation or call of module $Aj_r$ . It is necessary to say that a module $Aj_1$ is recursive and only if there is a cycle from $Aj_1$ to itself. Strictly speaking, $Aj_1$ is putatively [12, 18] since the necessary sequence of calls may never occur in any execution of S. We associate with each module $Ai_k$, for all i and k, a set of attributes in factual forms inserted automatically into a structure called a knowledge base.

Our approach is based on software engineering perspectives but built on top of the knowledge base that has capabilities for reasoning about objects. The knowledge base contains many facts and rules concerning entities that span the life cycle of a software development. The information contained in the knowledge base assists in determining the relationships amongst many objects. The deduced information and a set of transformation rules are used to generate a set of instructions to intelligently and interactively produce a complete software system. Software construction, we believe, can be more effective if the following facilities are provided:

- a method to identify modules and their relationships and dependencies,
- a method and criteria for version selection based on conditions and constraints, and
- a mechanism to determine the facts and rules that can be used to reason about objects and to assist in building a software system and/or sub-system.

The above facilities should provide information that is explicit and easily deduced and understood. This claim is consistent with Goguen's suggestion [13]: "*We need explicit information about how to put together the components of large systems, including which versions to use. Such information can be expressed in a module interconnection language and should be part of a program development environment*"

The proposed framework has advantages over other approaches. These advantages include:

- dependencies and relationships are automatically computed from the source code and stored in the knowledge base,
- support for multiple versions,
- suitable mechanisms for dynamic and static version selection,
- incorporation of the idea of sharing rules and definitions, and
- capabilities of reasoning and deduction.

There are three kinds of dependencies associated with each module version. These are:

1. module dependencies: where each version has its own list of imported dependencies,
2. logical dependencies: where each specification version has its own bodies and vice versa,
3. stub dependencies: where each body version has its own list of stub units.

All these dependencies are computed automatically from the source code and each version is stamped with the machine time to make it identifiable. Every time a target is specified the proposed approach automatically makes sure that all needed modules are up-to-date; otherwise the necessary action is taken to re-generate the required parts.

Most software construction tools provide a description of system structure and resource flow among modules. For example, Priento-Diaz and Neighbors [8] suggest a module description or syntax primitive. The proposed model provides Priento-Diaz's descriptions automatically. In addition, other information can be provided such as traceability of a module and module contents. Another advantage of our model compared with previous work is that it supports high-level descriptions of the relationships among modules. The underlying knowledge base provides sufficient information for a constructing tool to perform its job automatically which is unlike previous work.

For example, the description provided by [8] above solves part of the problem. Every time a software component is modified the description becomes out-of-date and it needs to be modified. This description, to be effective, requires an automatic aid or reminder to issue notifications of changes. In addition, the problem becomes worse if the software parts are written by different people and have complex interaction. The inclusion of multiple versions, which are inevitable, makes the problem more complicated.

In the following sections, information required for a software construction tool is described. This information, as mentioned above, includes dependency relationships, version selection criteria, and data and rules required to assist in building a software system out of predetermined components.

### 3.1 Dependency relationships

As mentioned earlier, a typical software system may consist of thousands of components. The relationships and dependencies among them can be difficult to

determine without automatic aids. These relationships, which describe the structural properties of a software system, are important to the success of software construction tools. They can be classified into:

- module dependencies (or "utilize" dependencies),
- logical dependencies ( or "realize" dependencies),
- stub dependencies, and

Others such as documents associated with each component or relationship with objects associated with other phases of the life cycle. The first three are automatically computed from the source code of the implemented language. They are discussed, in turn, in the following sections.

### 3.1.1 Module dependencies

In real life applications, a module may refer to other entities outside its visibility (i.e. imports entities from external modules). The visibility of these entities can be established in a number of ways depending on the implementation language. For example:

- Ada uses 'context-clauses', - C uses 'include', - Pascal uses 'extern' and 'units', ... etc.

Any compilation unit can have dependencies listed in front of ( or within) it. The proposed framework uses the compilation unit kind description as an attribute to distinguish the unit type. For example, dependencies, particularly the 'utilize' ones can prefix a specification part of a component, the body parts, and/or subunits. Each compilation unit has its own version number which determined by concatenating the unit name and the machine time stamp. Each version has its own list of dependent modules. This is represented by:

module_dependency (Version_No, List_Of_modules,Unit_Kind_Descript ion).

The argument Unit_Kind_Description is used to identify compilation units when they are submitted in one compilation.

### 3.1.2 Logical dependencies

Modern software engineering principles suggest that a software component should have two parts (a specification and a body) but most software construction tools have experienced difficulties in expressing multiple versions of interfaces and bodies. This drawback makes most available software construction tools impractical for real life

applications. The proposed framework offers a convenient way of allowing any interface to have any number of implementations. All implementation versions of a particular interface can be identified through the fact:

has_body (Interface_Version_No,Body_Version_No,Component_Name).

Using constraints and conditions one can choose between the alternative bodies for a particular specification.

### 3.1.3 Stub dependencies

Each version of a body may have different stubs (in the case of Ada) or may implement different program units or algorithms. when a body is analyzed all its program unit specifications are extracted and represented as facts in the form:

composed_of(Body_Version_No,List_Of_Program_Units,Body_Unit_Kind_Desciption).

Stubs can be found using the following rule:

```
Stubs( VersionNo, Unit_Kind_Description, StubsList):-
        composed_of( VersionNo, List, Unit_Unit_Description ),
        check_stub_and_return_stublist( List, StubList).
```

The automatic generation of dependencies is valuable for large scale projects. These projects usually contain thousands of modules and dependencies and the manual management of such dependencies is tedious and error-prone. The above information which is recorded in the knowledge base describes fully the interface control component of a system.

## 3.2 Selection of components

It is expected that a large number of components will be associated with a project of even a modest size. Each component exist in multiple versions where each version has its own characteristics or attributes to identify it such as:

time of creation, authorization, reliability, performance, debugging state, type of language (Ada, for example, supports components written in other languages), typeof operating system, history, reusability description and so on.

Version selection is an important aspect of software development but software construction tools do not give it much attention [8, 9]. The research in this chapter adopts a flexible approach in selecting the right version. This selection depends on the

user's own criteria or on selecting a default one that could be a 'preferred' or the 'latest' version. In fact, any policy or convention that can be expressed as a rule could be adopted. The selected version can be compiled and its status can be notified through the use of the fact:

compiled( Version_No).

This new status allows new versions of a unit to be represented without them being treated as recompilation units ( and hence replacing the version in use). A pre-set condition can also be used to govern the selection of the appropriate version for recompilation.

## 4. Compilation Strategies

A software system, in practice, may be decomposed into a large number of small sub-systems or components. These components need to be inter-connected automatically to produce the required system. Sufficient information needs to be provided with each component in order to effectively support software construction. Such information may include component's interfaces, location, exported entities, imported ones, number of versions, date of creation, component description, and traceability. When the above information is made available with each component the following two methods, which are required to accomplish components inter-connection, can be more effective:

- a method to identify the right version of the right component under the required conditions and constraints and to pull the right versions of the right dependencies,
- a method to formulate the programming language rules to guide the constructing tool.

A logic-based framework has been developed to compile a software system (an Ada program) according to the rules governing the dependencies of compilations. During a compilation, the system traces the compilation units submitted in order to perform the following steps:

* if the unit is in the program library then a recompiling action is performed(see order of recompilation below); otherwise perform the following step.
* if it is a library unit, but not a main, it will be compiled. If it compiled successfully it will be registered in the program library as a compiled unit. Although, it is irrelevant to the compiled library unit, the system will find

all its dependencies and compile them. This is needed when the main program is compiled.

* if the compilation unit is a secondary unit, the system will compile it, and find all its nested dependencies i.e. subunits.
* if the submitted compilation unit is a library unit and it is a main then the system will automatically find all its direct and indirect dependencies and compile them according to the Ada compilation rules, in order to ensure semantic consistency across unit boundaries.

Compilation dependencies between modules, without automatic generation, may make many aspects of maintenance hard to control. One example studied by Adams et al [14] in which they analyzed change logs for a new carefully engineered system, written in Ada, resulted in the conclusion that more than half of compilations were unnecessary.

### 4.1 Order of recompilation

When a compilation unit is modified and then recompiled, the whole Ada program, of which it is a component, does not have to be recompiled [15]. The units that need to be recompiled are identified according to the rules of recompilations:

* a library unit requires the recompilation of all the compilation units that depend upon(i.e. use) this library unit.
* if a compilation unit is not a library unit and consists of a package body or subprogram body, it only requires the recompilation of the subunits declared within its body. Other compilation units which use the modified compilation unit do not need to be recompiled, because they do not depend upon the implementation of the package or subprogram body.
* a subunit does not require the recompilation of a parent_unit or any other subunits.

A compilation unit stored in the program library is said to be up-to-date if it has been compiled more recently than the compilation units on which it depends and they are, in turn, up-to-date. Hence to ensure that a compilation unit is up-to-date it must be recompiled if any of the compilation units on which it depends have been modified. A compilation is parsed only once and there is no need to parse it again when it is recompiled because all the required information concerning its compilation units have been extracted and registered in the program library.

When a unit is submitted to a compiler, the proposed system will search the program library for it. If it does exist then the system will take a recompilation action,

otherwise it will take compiling action. For the recompilation action, the system will automatically recompile all the necessary units according to the Ada recompilation rules.

## 5. Automatic Building of Software Systems

A version can be chosen according to a set of conditions. These conditions can be viewed as a set of descriptions required to determine the right version. These descriptions can take any form. For example: find the version that is: *reliable, written in Ada, and created after May*15, 1995. Software construction assistant can be invoked using commands of the form:

*acm module_name.*

We use 'module_name' as the argument instead of the file name since we believe that programers and users prefer to think of modules and packages rather than files and directories [16]. The proposed construction assistant determines the status of the module ( i.e. compiled or uncompiled). When the status is uncompiled a compilation action is taken otherwise recompilation is performed. The proposed system also determines the type of the submitted module, i.e. specification, body, or a subunit.

Once the type has been determined the system selects the version that satisfies the preset conditions. When a version has been chosen , the system check its state (i.e. is it already compiled). If the unit is not previously compiled, then its with-clause dependencies will be traced, where a version is linked to its with-clause through the fact:

with_clauses (VersionNumber, ListofWithClauses, Unit_Kind_Descrption).

where a Unit_Kind_Description is a term in the form:

subprogram_specification (UnitName). subprogram_body ( UnitName).
package_specification (UnitName). package_body ( UnitName).
generic_specification (UnitName). generic_body ( UnitName).
task_specification (UnitName). task_body (UnitName).

The Unit_Kind_Description is essential if a specification and its body were submitted as one compilation. When a unit is submitted to the constructing tool in the form:

*acm module_name.*

the system identifies whether it is a compilation or a recompilation. Once it has been

identified as a recompilation the proposed system determines its type ( i.e. a specification, body, or a subunit). A version is then selected according to preset conditions.

When the system can not find a version that satisfies the preset conditions then a message is generated to inform the user that there is no such version. At the same time information concerning a default version is displayed (such as version number, unit description, file name, and so on). The system asks the user for permission to choose the default one. If permission is given then the system proceeds in its job, otherwise compilation is terminated. All modules which are affected by the submitted module are identified using:

*outofdate (Module_Name, List_Of_Aff_Modules).*

Necessary recompilations are performed according to the language order of recompilation. Any unit can be submitted for compilation or recompilation without affecting system consistency. Unlike the PCTE and PCTE+[17] interfaces the current implementation has facilities to support querying the consequences of changing a module.

## 6. Module Invocation Algorithm

The set of invoked modules R, that belong to the graph G= (V, E), can be generated using the following proposed algorithm.

### 6.1 Module generation algorithm

The proposed algorithm is simple and easy. It identifies initial conditions and the main loop. It is constructed as follows:

**initial condition**

R:= {S};
Total-Set := $\{Ai_k\}$; (* for all i and k *)

**main loop**

repeat T:= $\phi$; (* T is a temporary set *)

```
        for all Aj_r in total-set do
        T: = T U { w | (Aj_r, w) is in E };
        Total-Set := T - R ;          (*    The new reachable nodes    *)
        R : = R U Total-Set
until Total-Set = ϕ;
```

On exit, we expect the algorithm to generate all reachable modules in graph G = ( V, E).

### 6.2 Module invocation cost

Invocation (or calls) is, simply, a permutation of V modules where each module may be invoked zero, one, or more times.

$$\sum_{j=1}^{M} calls\ (Ai_k,\ Aj_r)$$

where 1 <= i, k, j, r <= V, M = (v -1) + Q, and Q is the total extra calls for all modules except the driver S. To compute the total cost, an estimation of the time required for each call, on average, can be calculated. The estimated cost of a call includes searching for the right module's version, i.e. the version that satisfies the imposed conditions set up by users or even the default conditions. Suppose the estimated cost for a call is $t_{call}$ , the total cost for building the proposed system is:

$$t_{call} \times \sum_{j=1}^{M} calls\ (Ai_k,\ Aj_r)$$

The complexity of the algorithm depends on the size of software applications being applied.

## 7. Working Examples

This section describes the tools and methods used to build a software system or sub-system out of pre-tested modules (i.e. compile and recompile a software system automatically). The tool for constructing software modules can be invoked using commands of the form:

```
acm     system_drive.
```

The system checks whether the component 'system_drive' is submitted for compilation or recompilation and passes control appropriately. When the program is parsed the following items of information are extracted.

```
file( '/fs/cs/abid/C_M/ADA_MANAGEMENT/direct_io', direct_io,
generic_specification( direct_io), direct_io7689054).

file( '/fs/cs/abid/C_M/ADA_MANAGEMENT/direct_io', direct_io,
package_body( direct_io), direct_io7689054).

with_clauses(direct_io7689054, [io_exceptions, system],
generic_specification( direct_io)).

with_clauses(direct_io7689054, [stdio, unix],
package_body( direct_io)).
```

The above facts state that a unit 'direct_io' has two parts, a specification which has version 'direct_io7689054' and a list of dependencies [io_exceptions, system] and a body which has version 'direct_io7689054' and a list of dependencies [stdio, unix]. The schema of contents consists of three attributes: version number, unit kind description, and the list of constituents. Other items are also extracted such as:

```
package_specification(decl).
package_specification('Complex_Relations').
package_specification('Manipulate_Vectors').
package_specification('Iterate_Swan_Method').
package_body('Complex_Relations').
package_body('Manipulate_Vectors').
package_body('Iterate_Swan_Method').

sub_unit(subprogram_body(divide)).
sub_unit(subprogram_body(read_in)).

with_clauses('Complex_Relations90123103128',[decl,text_io,'MATHS_LIBRARY'],
                    package_specification('Complex_Relations')).

with_clauses('Manipulate_Vectors90123104043',[decl,text_IO],package_specification
                    ('Manipulate_Vectors')).
```

```
with_clauses('Rotate_Axes90123104628',[text_IO,'MATHS_LIBRARY','MATHS_
CONSTANTS',decl,'Complex_Relations','Manipulate_Vectors'],package_specificatio
        ('Rotate_Axes')).

parent_unit(['Complex_Relations'],sub_unit(subprogram_body(divide))).
parent_unit(['Complex_Relations'],sub_unit(subprogram_body(read_in))).
parent_unit(['Complex_Relations'],sub_unit(subprogram_body(scalmult))).

reusable(package_body('Iterate_Swan_Method'),'Iterate_Swan_Method90123111815
        ',[ describes,implementations,'Swan',methods],re_usable).
reusable(package_specification('Iterate_Swan_Method'),'Iterate_Swan_
        Method90123111637',[specification,'Swan',algorithms],re_usable).

reusable(package_body('Refine_Interpolation_Fit'),'Refine_Interpolation_
        Fit90123111441',[implementation,interpolation,function],re_usable).

reusable(package_specification('Refine_Interpolation_Fit'),'Refine_
        Interpolation_Fit90123111320',[specification,interpolation],re_usable).
```

The above sample of facts is useful for a number of purposes such as building systems, reusing components of a system, and/or querying the knowledge base itself.

### 7.1 Example on automatic compilation

This example demonstrates how the tool can integrate a large number of modules with multiple versions. It shows also how the system interacts with the user. Firstly the compilation unit 'optimize', which is considered as the driver of the system S, is submitted to the integrating tool as follows:

```
acm optimize.
```

The constructing tool finds the 'optimize' version which matches the user requirements. If there is no version matching the pre-set requirements a message is issued as follows:

```
Condition Set Is Not Applicable For subprogram_body(optimize) Which has
Version optimise90128162412 And It Is In File b28162412.A PLEASE MAKE UP
YOUR MIND !.
Would you like to choose a default version? y
```

When a user accepts the default version mentioned above by typing yes(y) the tool

proceeds to find first all the version dependencies. For example the version 'optimize . . .' depends on 'decl', ' Complex_Relations', 'text_IO', 'MATHS_LIBRARY','MATHS_CONSTANT','Manipulate_Vectors','Iterate_Swan_Method', and 'Rotate_Axes'. The tool automatically extracts the dependencies. It applies the user requirements on each of the dependency versions respectively as shown below:

Condition Set Is Not Applicable For package_specification(decl) Which has Version decl90128154431 And It Is In File s28154431.H PLEASE MAKE UP YOUR MIND!. Would you like to choose a default version? y

Condition Set Is Not Applicable For package_body(Iterate_Swan_Method) Which has Version Iterate_Swan_Method9012816231 And It Is In File b2816231.A PLEASE MAKE UP YOUR MIND !.
Waiting for the root of the system optimise90128162412 to be compiled:
Exit from Command execution.
yes

As can be seen from above, the integrating tool analyses component dependencies and matches qualities required by a user with qualities associated with each version of the component. When there is no matching version available the system reports the situation to the user; otherwise the system submit the required version to the compiler manager.

### 7.2 Example on automatic recompilation

When a component is submitted for a recompilation, the type of the component is determined and control accordingly passes to one of the following procedures:

- recompile_specification,
- recompile_body, or
- recompile_subunit.

Otherwise a message is generated to draw the users attention to an illegal action. When the 'recompile_specification' procedure is taken the system recompiles the specification; recompiles its body and subsequentlythe related subunits. It then collects all components that were made out of date because of the recompiling of the above specification and recompiles them accordingly.

When control passes to the 'recompile_body' procedure the system recompiles the body; recompiles all its subunits. It then collects all the components made out of date

because of the recompiling of the above body ( if the body is a library unit) and recompiles them. A body does not require the compilation of its specification.

When control passes to the procedure 'recompile_subunit' the system recompiles the submitted subunit only (i.e. a subunit does not require the recompilation of a parent-unit or any other subunits). The following examples show a number of test data files. The example demonstrates how the system recompiles all modules made out of date when a specific module was recompiled as shown below:

acm 'Complex_Relations'

The system checks first that the submitted module was already compiled, hence, it assumes that the module was submitted for recompilation. As a result a number of other modules will be made out of date. Hence the system checks all related dependencies and proceeds by applying the user requirements on each version as shown below:

*Condition Set Is Not Applicable For package_specification(Rotate_Axes) Which has Version Rotate_Axes9512816743 And It Is In File s2816743.H*
*PLEASE MAKEUP YOUR MIND !.*
*Would you like to choose a default version? y*
*Would you like to choose a default version? y*
*Waiting to compile package_body(Iterate_Swan_Method) which is in file b2816231.A and has Version No: Iterate_Swan_Method9012816231*
*Exit from Command execution.*

*Condition Set Is Not Applicable For subprogram_body(optimize) Which has Version optimise95128162412 And It Is In File b28162412.A PLEASE MAKE UP YOUR MIND !.*
*Would you like to choose a default version? y*
*Waiting for the root of the system optimise90128162412 to be compiled:*
*yes*

The constructing framework, as mentioned above, checks version numbers to establish whether a version was submitted for compiling or recompiling. This information was deduced from the program library where the status of a component was stored. This facility provides a way of compile new versions without interpreting them as a recompilation unit.

The test data used has over a 100 compilation units distributed over 30 source files totaling about 4000 lines. If, for example, a request is issued to compile a main unit then the compilation of all the required units is handled automatically according to

the order of compilation and recompilation. The experimental prototype for software construction provides users with flexible techniques for building software systems or sub-systems. These techniques support automatic selection of versions of components according to a specific set of requirements. They also support automatic traceability of direct and/or indirect dependencies of a component and they support automatic building.

The result of the experiments show how the knowledge base facts and rules can be used by the system itself to deduce new information and to support queries about component and their relationship. The results also show how version selection criteria can be used according to local needs. The selection combines both static and dynamic selections. This facility is not found in previous work including PCTE(+) [17].

## 8. Conclusions and Future Work

A software integration tool is one that specifies the structural properties of a software system. A software system can be constructed when at least the following methods are available:

- a method to establish dependency relationships,
- a method for version selection, and
- rules for constructing a piece of software.

The proposed framework provides automatic computation of compilation and recompilation dependencies. It also provides a way to compile new versions without interpreting them as recompilation units. Constructing a software system from a library of components has a great influence on the promotion of software reuse.

Although the proposed framework has facilities for Object-Oriented techniques, we suggest to extended future work to incorporate more facilities and methods for Object-Oreinted and/or Artificial Neural Network as both techniques appear to be promising towards enhancing software methodology.

## 9. References

[1] Jazaa, A. "Toward Better Software Automation." ACM SIGSOFT, *Software Engineering Notes,* 20, No. 1 (January 1995),79-84.

[2] Corssini, P. and Lopriore, L. "An Implementation of Storage Management in

Capability Environments." *Software-Practice and Experience,* 25, No. 5 (May 1995), 501-520.

[3] DeRemer, F. and Kron, H. " Programming-in-the-Large Versus Programming-in- the-Small." *IEEE Transactions on Software Engineering,* (June 1976) , 321-327.

[4] *The STARTS Guide.* NCC Publication, Second Edition, 1987.

[5] Narayanaswamy, K. and Scacchi,W. "Maintaining Configurations of Evolving Software Systems." *IEEE Transaction on Software Engineering,* 13, 3 (1987).

[6] Baalbergen, E. H.; Verstoep, K., and Tanenbaum, A. S. "On the Design of the Amoeba Configuration Manager." *Proceedings of the 2nd International Workshop on Software Configuration Management,* ACM SIGSOFT Software Engineering Notes, 17,(7) (1989), 15-22.

[7] Prieto-Diaz, R. and Neighbors, J. M., *Module Interconnection Languages: A Survey.* U. Calif., Irvine , ICS Tech. Report 189 (August 1982).

[8] Prieto-Diaz, R. and Neighbors, J. M. "Module Interconnection Languages." *The Journal of Systems and Software,* 6, No. 4 (November 1986), 307-334.

[9] Tichy, W.F. "Tools for Software Configuration Management." *Proceedings of the International Workshop on Software Version and Configuration Control,* Teubner Verlag, Stuttgart, FRG (January 1988), 1-20.

[10] Sully, P. *Modelling the World with Objects.* Prentice Hall, 1993.

[11] Loy, P."The Method Won't Save You", ACM SIGSOFT, *Software Engineering Notes,* 18, No. 1(January 1993), 30-34.

[12] Lam,S. S. and Udaya, A. "A Theory of Interfaces and Modules I-Composition Teorem." *IEEE Transactions on Software Engineering,* 20, No. 1 (January 1994), 55-71.

[13] Goguen, J. A. "Reusing and Interconnecting Software Components." COMPUTER, IEEE Publication (February 1986), 16-28.

[14] Adams, R.; Weinert,A., and Tichy,W. "Software Change or Half of All Ada Compilations are Redundant ." *European Software Engineering Conference,* (1989).

[15] Hitchcock, P. "A Database view of the PCTE and ASPECT." In: *Software Engineering Environment,* P. Brereton (ed.), Ellis Horwood (1988), 37-49.

[16] Jordan, M. "Experience in Configuration Management for Modula 2." *Proceedings of the 2nd International Workshop on Software Configuration Management,* ACM SIGSOFT Software Engineering Notes, 17, No.7 (November 1989), 126-128.

[17] Commission of the European Communities, ESPRIT Program, "Rationale for the Change between PCTE+(issue 3) and PCTE (Version 1.5)." *IEPG TA- 13,* January 6, 1989 (PCTE+/RAT/03).

[18] Babel, L. "Isomorphism of Chordal Graphs", *Computing,* 54, No. 4 (1995), 303-316.

# إطار عام مبني على المنطق لتصميم نظم البرامج

**عابد جازا**

*قسم علوم الحاسب، جامعة اليرموك، اربد، الأردن*

**ملخص البحث** . تصف هذه الورقة إطارًا عامًا مبنيًا على المنطق لبناء برامج متناسقة من عدة وحدات. تمت ترجمتها كل على حدة. الطريقة المقترحة تتصرف كآلية ربط بين المكونات المختلفة وتتعامل مع العديد من العناصر المتعلقة بالوحدات المكونة للبرنامج. هذه العناصر يتم استخلاصها في الواقع من نصوص البرامج. تسمح الطريقة أيضًا بتعدد النسخ والاختيارات الثابتة والديناميكية. كما تسمح الطريقة المقترحة بإظهار أشكال الوحدات المكونة ومتعلقاتها.