An Interconnection Language for Reuse at the Template/Module Level

M. K. Zand and H. Saiedian
Department of Math and Computer Science, University of Nebraska at Omaha, Omaha, Nebraska

K. M. George and M. H. Samadzadeh
Computer Science Department, Oklahoma State University, Stillwater, Oklahoma

Reuse is becoming a potent trend in software development and a major way to boost software productivity. To put reusable software units together, one has to be able to find them with minimal effort in the first place. The effort needed to access, understand, and customize the code must be less than the effort required to create new code. A simple library of components cannot provide sufficient methods to facilitate the selection and interconnection of the reusable modules. The context of this work is the ROPCO (reuse of persistent code and object code) environment and the primary candidates for reuse are the modules and templates. The objective of this article is to present the design of an interconnection language which can be incorporated with other ROPCO components to facilitate the selection, customization, and interconnection of reusable modules in the ROPCO software development environment. This language helps to define the interface specifications of the components and find the best module(s)/template(s) meeting the desired specification. The detailed algorithms of the operations that are necessary at the user level to support the reuse of available components are given and described in detail with a view toward verification.

1. INTRODUCTION

An interconnection language is a machine-processable specification language that provides the means for a system designer to represent the system in a concise, precise, and verifiable form. Once the system structure has been designed, it can be coded in an interconnection language. This code is used to verify the design completeness and detect inconsistencies before the system is actually linked together.

Conventional module interconnection languages (MILs) provide formal grammar constructs (or other types of formalisms) to specify and eventually assemble a complete software system [1]. Automatic processing of such formal specifications requires system-integrity and intermodule-compatibility verification. Conventional MILs are not concerned with the specification information (i.e., the actual function of the system) or the detailed design information (i.e., how the modules implement their functions). Typically, they are primarily concerned with architectural design information (i.e., how the modules fit together). Prieto-Diaz and Neighbors [1] consider the following items as the main characteristics of an MIL:

1. A separate language is used to describe the system design.
2. It can perform static type checking at an intermodule level of description.
3. It can consolidate the design and construction processes (i.e., module assembly) into a single description.
4. It can control different versions and families of a system.

Tichy and Baker [2] and Perry [3, 4] emphasize the close relationship that exists among the MIL, the interface control system, and the version control system in a software development environment.

Goguen's LIL (library interconnection language) [5] and Perry's INSCAPE [6, 7] use most of the ideas

Address correspondence to Prof. Mansour Zand, Department of Math and Computer Science, University of Nebraska, Omaha, NE 68182.

655 Avenue of the Americas, New York, NY 10010
0164-1212/93/$6.00

found in conventional MILs such as INTERCOL [8] and GANDALF [9]. However, they go beyond these MILs by incorporating tools for specifying the semantics of the modules.

The general context of this work is the ROPCO (reuse of persistent code and object code) environment [10]. ROPCO's module/template interconnection language (RMTIL) is not a stand-alone "tool"; rather, it is a part of the ROPCO environment specifically designed to utilize and integrate with the other components of ROPCO. Section 2 covers ROPCO by providing a general overview and more detailed description of the components that are directly related to the discussion of the interconnection language. This brief introduction to ROPCO is deemed necessary because an understanding of the ROPCO environment will help clarify and justify a number of the design decisions and implementation choices.

This article is organized into seven sections. Section 2 is a brief introduction to the ROPCO environment and its design goals. Section 3 is a survey of existing interconnection models and Section 4 introduces ROPCO's interconnection model and language. Sections 5 and 6 discuss interconnection operations and processes. A description of the relevant operations as well as verification conditions are discussed. The last section summarizes the work and gives a progress report on the implementation of the ROPCO environment.

2. ROPCO

Reuse is becoming a potent trend in software development and a major way to boost software productivity. Several alternatives have been proposed for the level of software reuse. They include specification level, design level, program/subprogram level, module/template level, and code and object code level [1, 11, 12]. Specification and design are at higher levels of abstraction than the other reuse levels; therefore, their potential for accommodating reuse is greater and adaptation to new applications can be expected to be simpler. However, the reuse process of the specification and design levels ultimately involves coding (either manually or automatically), testing, and debugging. On the other hand, reuse at the code level reduces the coding and overall testing efforts; hence, one can argue that it is more economical.

The result of one survey shows that 40–60% of all code is reusable from one application to another, 60% of the design and code on all business applications is reusable, and 75% of program functions are common to more than one program [13]. The survey also indicated that only 15% of the code found in most programs is unique and novel to each specific application.

The results of an experiment show that the optimal size of a reusable module is about 200 to 250 code lines [14]. Reuse of such modules is usually incremental, i.e., it results in altering only one portion of the code of a system. In most of the existing reuse methods, older versions of the resulting module at each step either are not saved at all or are not stored efficiently. It turns out that there is no need to store the entire module under a new version number or recompile the whole module. The modified portion of the code is normally one or more of the constructs of the language (referred to here as a template); consequently, it is natural to consider templates as units of reuse and store only those templates that are modified or are added to a module [15].

ROPCO is a software development environment that adopts a novel approach to reuse utilizing code and object code stored in persistent structures [10]. ROPCO was designed to accomplish the following goals:

1. Identify and provide access to a segment of code and the related object code based on user specifications.
2. Facilitate module modification both at the programmer and compiler levels.
3. Provide facilities to efficiently store and retrieve all versions of the module(s) under consideration.
4. Facilitate module/template inter- and intraconnection.

Minimizing the user/programmer efforts spent on tasks other than programming was one of the main motivations in the design of the ROPCO system. We emphasized minimizing the compilation time and effort. ROPCO's minimal compilation strategy limits the propagation of microincremental compilation resulting from intraprocedure modifications, while consistency of variables and the intramodule flow graph remain intact. The problem of macroincremental compilation resulting from interprocedural modifications is addressed in ROPCO's design [16].

In ROPCO, templates/modules are chosen as units of reuse and the novel concept of a "use network" is devised [10]. Another design goal of ROPCO was to furnish the system with a pragmatic schema for storage of versions of a module and maintain direct and sequential access to the blocks of modules. Hierarchical and flat persistency methods were used to achieve this goal.

ROPCO consists of three main subsystems: identification mechanism, software control mechanism, and interface. Figure 1 illustrates the main components of ROPCO and the relationships among them. The following is a brief description of the main subsystems of ROPCO.

2.1 Definitions

The following definitions are provided to clarify the presentation.

Template: A segment of code corresponding to a syntactic construct of a high-level programming language.

Basic template: A template that contains straight-line code or a loop or conditional statement which contains only straight-line code (templates T1 and T3 in Figure 2 are examples of basic templates).

Composite template: A nonbasic template is called a composite template (template T2 in Figure 2 is an example of a composite template).

Outermost template: A basic or composite template that is not embedded in other template(s) (templates T1, T2, T3, and T4 are examples of outermost templates in Figure 2).

In the rest of this article, the three terms name, object, and identifier are used interchangeably.

2.2 Identification Mechanism

The identification mechanism (IDM) is designed to identify and select one or more module(s)/

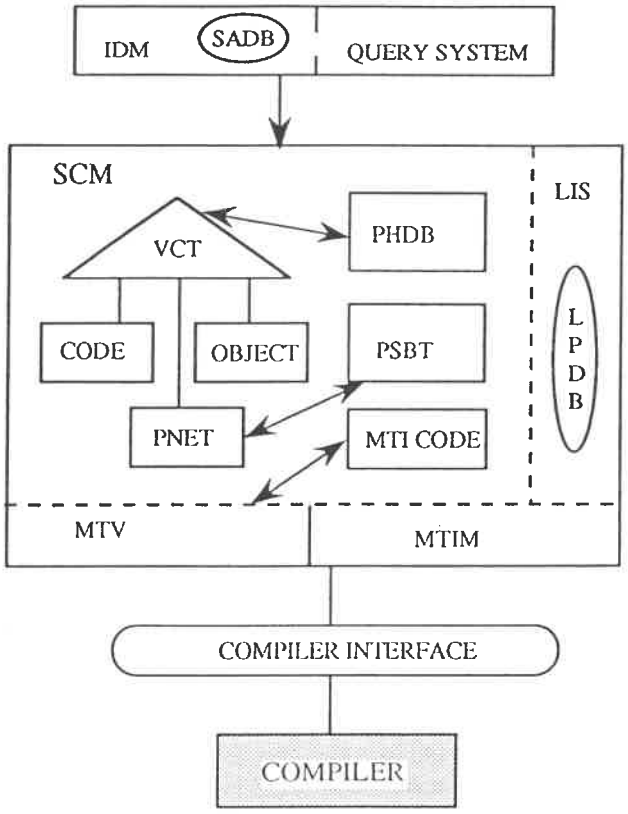

Figure 1. Main components of ROPCO.

IDM:	IDentification Mechanism	LIS:	LInkage System
LPDB:	LIS Persistent DataBase	MTIM:	Module/Template Inter-connection Mechanism
MTV:	Module/Template Verifier	PNET:	Persistent NETwork
PHDB:	Persistent History DataBase	PSBT:	Persistent SymBol Table
SADB:	Software Attriburt DataBase	SCM:	Software Control Mechanism
VCT:	Version Control Tree		

```
        MODULE Example

        BEGIN
T1          i := 0
T2          DO WHILE  i < n
            BEGIN
T2.1            READ(cond)
T2.2            IF cond =1 THEN
                    i := i + 2
                ELSE
                    i := i + 1
                END IF
T2.3            PRINT (i)
            END WHILE
T3          y := prod * x
            y := (y + i) * x
            y := y + c
T4          IF y < 0  THEN
                y := 1
        END {Example}
```

Figure 2. A hypothetical module.

template(s) that meet user requirements. Each module is classified based on its function, environment, and implementation attributes. The attributes (which are stored in the software attribute data base (SADB)) are used to identify the candidate module(s).

2.3 Software Control Mechanism

The software control mechanism (SCM) is the kernel of ROPCO. The main tasks of SCM include arranging access to different versions of a module (at module and template levels), coordinating the compilation and recompilation tasks, and maintaining variable consistency throughout the modules (and the system). The main components of SCM are introduced in the following subsections.

2.3.1 Program history data base (PHDB). This component of SCM is designed to retain the information required to access the root of the family tree of a module/program and to hold information on how to move from a specific version to the successor or predecessor versions. To access the family tree (or version tree) of a module, the header pointing to the root of the version tree can be found in the module record in PHDB. To move from one version to another, information on incremental changes to each version is also deposited in the module record of the working module.

2.3.2 Version control tree (VCT). VCT is constructed based on a persistent B-Tree [10]. Internal nodes of the VCT serve as indices and its external nodes contain addresses of the clusters that hold code as well as a header for the chain of variables used in each template. Each internal node has k sets of child pointers and each pointer has a time stamp to indicate the version number. Each external node contains the address items related to the template represented by the external node. The external nodes are linked to maintain sequential access to the clusters; this is in addition to the direct access provided to the template clusters.

2.3.3 Persistent symbol table (PSBT). The symbol table of a persistent programming environment for a given program exists as long as the program itself exists. The life cycle of the symbol table of a program does not terminate at the end of the execution of the program. The symbol table of a persistent program should be able to support all versions of the program. The main difference between this type of persistent symbol table and the conventional symbol table is that, in PSBT, one variable may be defined in more than one context and thus can have different types. However, only one type of variable is allowed in each instance (version) of the table. Each variable can have one or more records in the table.

2.3.4 Persistent network (PNET). PNET is designed to facilitate the name-space consistency throughout the system. Each unit of program which is allowed to declare variables has one persistent symbol table. Each template of a program has a chain of pointers to the symbol table—one for each variable used in the template. Each variable in the symbol table has a chain of its uses in different templates. The two sets of chains for all variables and templates constitute the "use network" (i.e., PNET). When a declaration is inserted into or deleted from a template, the use chains are searched to access the relevant variables in the template.

2.3.5 Module / template verifier (MTV). MTV is designed to check for compatibility of interconnection/substitution of modules and templates. This subsystem of ROPCO uses an automatic verifier and a semantic specification model to verify functional consistency among the interconnecting components.

2.3.6 Module / template interconnection mechanism (MTIM). MTIM is the subsystem responsible for interconnection and substitution of modules/templates. MTIM uses other SCM components to perform its task.

2.4 Interface

The interface is designed to facilitate microincremental compilation between ROPCO and the language-specific compilers. It prepares the templates/modules that are altered or otherwise affected by modification for compilation. As a postcompiling activity, the interface transfers the address of the newly created object code and symbol table to SCM.

2.5 Access and Update

The unit of access and update in ROPCO is the template. Code and object code for each template are stored in separate areas of secondary storage. Once a specific version of a module is recalled for reuse or modification, VCT is loaded into the primary memory and blocks of code and object code are accessed and loaded by traversing the external nodes of the given version of the module. Next, the PSBT of the modules under consideration is loaded into the memory to provide the current version of the symbol table.

3. INTERCONNECTION MODELS

Perry [3] defines an interconnection model (IM) as an ordered pair consisting of a set of objects and a set of relations which represent the interconnections that exist among the objects:

IM = ({objects}, {relations})

An IM is used to construct a graphic representation for a system where objects are nodes and relations are edges [17, 18].

Perry classified IMs into three categories. The following sections present these three classes.

3.1 Unit Interconnection Model

A unit interconnection model (UIM) states the relationships that exist among components of a software system (basically files and programs). The nature of the relationships among these units is in the form of dependency of one unit on some other unit(s). This can be represented by the following ordered pair [4]:

UIM = ({units}, {"depends on"})

The main advantage of this model is that it promotes the notion of encapsulation and persistency for units through modular construction of software [3]. This model could be used to determine compilation setting, recompilation strategy, and change notification.

IMPORT in MODULA-2, **#include** in C, and **with** in Ada are examples of features in programming languages that facilitate compilation setting using the UIM approach. Other examples of the application of UIM are Make and Software Code Control System (SCCS) [19]. An extension of SCCS uses the "depends on" relation information to inform the user of the effect of a change throughout the software system. Cedar's system modeler [20] and the DOMAIN software engineering environment [21] use UIM in their environments.

While UIMs are practical in supporting modularity, compilation, recompilation, and change notification, they are applicable only to module-level interconnection and large-grained objects. They cannot support fine-grained object interconnections, compilation, recompilation, or change notification.

3.2 Syntactic Interconnection Model

A syntactic interconnection model (SYIM) is an ordered pair of a set of objects and a set of relations. SYIM focuses on the interconnections of objects used in the creation of a software system. Furthermore, SYIM indicates the changes that have occurred in the syntactic objects. An SYIM has the following form [4]:

SYIM = ({functions, procedures, types, variables, ...},
{"is used at", "is set at", "calls', ...})

The objects set of an SYIM consists of procedures, functions, types, and variables. The relations set of an SYIM consists of the relationships existing among its components or objects. The list of objects and their relationships are language dependent (e.g., some languages support import and export, and some do not). The main advantage of SYIM over UIM is its ability to support the fine-grained units or components of the software system and "provide relations among the syntactic elements of a programming language" [4].

SYIM could be used for change management, smart recompilation, and system modeling. The cross reference generated by the compilers can be used to detect the extent of the impact of the changes made on the syntactic units [11]. For example, Interlisp MasterScope [22] has a data base for relationships among objects; change management is provided through querying the data base.

Tichy [17] uses a syntactic model in Smart Recompilation. The effect of changes on the syntactic objects within the model could be determined using SYIM. In GANDALF's system composition and version control for Ada environment [12], system modeling is mentioned as another example of the application of fine-grained IM.

3.3 Semantic Interconnection Model

The SYIM approach doesn't provide any information as to why the interconnections exist and how the objects are intended to be used. The lack of semantic information makes it difficult to provide effective management of system evolution. To facilitate the building of a large software system from small components, we need capabilities to represent the purpose of the objects and how these objects are to be used. Formal specification of objects is a powerful tool for providing semantic information [3].

Although algebraic specification is perhaps more suitable for describing the relationships among operations and expressing the purpose of the operations, Hoare's input/output predicates are more meaningful in representing the interconnection points and proving properties about a program fragment. A semantic interconnection model (SEIM) has the following form [4]:

SEIM = ({functions, procedures, types, . . . , predicates},

{"is used at", "is set at", "calls", . . . , "satisfies"})

The SEIM is used in the design of the module interface specification of INSCAPE.

4. ROPCO INTERCONNECTION MODEL AND LANGUAGE

In the ROPCO environment it is not possible to use a conventional MIL. Rather, we need a language that provides capabilities to facilitate intraconnection of modules (i.e., substituting a template in a module, assembling a number of templates within a module, or creating a module as well as interconnecting a number of modules).

Using templates as units of reuse generally makes the "gluing" process more complicated than it would be if modules were used as units of reuse. This is because local variables are not involved in interconnecting modules, while they must be considered in interconnecting templates. This added complication makes the design of an interconnection language more difficult. However, this is a price to be paid for potentially reducing the scope of recompilations and for the availability of the microcompilation facility in the software development environment. Since module/template is selected as the unit of reuse in ROPCO, this type of language is called M/TIL (module/template interconnection language). To the best of our knowledge, to date no M/TIL has been reported in the open literature. Throughout the rest of this article we will focus on the specification of the template-related portion of M/TIL.

The functions of M/TIL are transparent to a programmer (user) who is concerned with the development of a software system using reusable components. The task of M/TIL is to provide the syntax and semantics of modules/templates and to be used to verify the consistency of the name space and the functional integrity of a module after a template replacement/insertion in the module or after the creation of the module by assembling a number of templates. Therefore, an M/TIL doesn't necessarily need to be self-expressive or easy to understand. However, the environment should incorporate tools such as a query language or graphic representation, which are some of the typical tools used for ordinary user interfaces.

As mentioned earlier, some of the conventional MILs are not concerned with the implementation specification; nevertheless, M/TIL uses the implementation specification for the verification process. The following sections provide the characteristics and functions of ROPCO's M/TIL (RMTIL) and the verification process of M/TIL.

4.1 M/TILCharacteristics and Functions

An M/TIL should have the following characteristics:

1. It should be a machine-processable and formal specification language.
2. It should contain verification conditions for interaction with an automatic verifier.
3. It should contain features to enable it to interact with the software environment to use the relevant information about the modules/templates.
4. It should be designed to provide the syntax and semantics of the modules/templates.

An M/TIL should have the following capabilities:

1. It should provide explicit information about how to put the modules/templates together.
2. It should provide the capability for microincremental modification verification.
3. It should provide the ability to manage different versions of a module/template.
4. It should provide the capability for semantic verification by an automatic verifier.
5. It should provide facilities for consolidating the structure and capabilities of the software environment for module/template assembly.
6. It should provide interconnection and intraconnection commands among and within the modules.

RMTIL consolidates other development/mainte-

nance components of ROPCO and uses the SCM structures. Type checking for name-space consistency[1] is not part of an automatic verifier; RMTIL uses persistent symbol table and persistent use network to perform this task. Therefore, the verifier's task is limited to checking for soundness of the semantics of the replaced template (or assembled templates) without worrying about type checking.

4.2 Module /Template Specification

This section presents the specification of an RMTIL module. An RMTIL module provides four specifications for each software module that it represents, as outlined below:

1. attribute specification, which includes environment, function, and version and family information;
2. module structure, which is the control flow information;
3. module interconnection specification;
4. template specification, which consists of template pre- and postconditions, other verification conditions, and the structure of the template (template control flow), if the template is a composite.

Perry's approach to the design of an interconnection language provides adequate power for module interconnection (i.e., the SEIM part) [4] and is used for the design and development of the MIL portion of our M/TIL. The following section presents the RMTIL model constructed based on the specifications given in this section.

4.3 The RMTIL Model

A typical M/TIL needs to provide semantic as well as syntactic information. Since other components of ROPCO are responsible for the task of syntax consistency of the module involved, the RMTIL model is concerned with semantic information only. The RMTIL model is defined below.

RMTIL Model = ({objects}, {relations})

and

object = (M, A, C, P, T), relation = S,

where

M is a module name

A is a set of module attributes include functional, environment, and implementation attributes [16]

C is the module control flow represented in terms of templates

P is the module pre- and postconditions analogous to Hoare's input/output predicates and modify set, which is list of all input objects modified by the module

T is a set of one or more templates t in the module

S is a set of properties such as "language is," "version is," "satisfies," "requires," "flow is," etc.

A template t is defined as a tuple:

$t = (\{id_t, a_t, p_t, c_t\}, \{s_t\})$

where

id is the template identifier

a is the set of template attributes (e.g., function, main objects, and version information)

p, c, s are predicate conditions, control flow, and the set of properties for the template, respectively.

This model provides all the information required for interconnection as well as intraconnection and version control. If the implementation is at the module level, then C and T may be empty. Furthermore, not all of the template attributes are required for interconnection and verification. The function and main objects attributes are used in the template selection process and are stored in SADB. These attribute are not required in RMTIL's specification of a module.

4.4 The RMTIL Language

The RMTIL language incorporates some of the concepts put forth in the design of CLU [23]. CLU was designed by Liskov et al. [24] in 1977 to implement the concept of abstract data types. CLU has a rich set of assertions for the module specification and verification process, which are used in RMTIL. To provide version handling, template specification, and control flow and to enrich module specification, the requisite features are added to the RMTIL. The following features have been added to RMTIL.

ATTRIBUTE is a structure assertion to provide environment information for the module.

The assertion symbols within the ATTRIBUTE are the same as those for the module attribute defined as A in subsection 4.4.

Template identifier and *version number* are added for version control.

[1] Name-space consistency refers to checking for the consistency of "names," "objects," or "variables" within the scope of the module(s) under consideration.

CONTROL-FLOW and *TEMP-CONTROL-FLOW* structure assertions are added to represent the flow of the program.

CONTROL-FLOW represents module control flow and has the following form:

CONTROL-FLOW: $(t_1, t_2, \ldots, t_n)$,

where $t_1, t_2, \ldots, t_n$ are outermost templates in the module.

TEMP-CONTROL-FLOW represents the structure of a composite template. Figure 3 illustrates various control flow representations in RMTIL.

An RMTIL specification module consists of a module header and a module body. The module header, called MODULESP, contains the module attribute, verification predicates, and control flow. The body of a module, called TEMPLATESP, contains the specification for one or more templates. A formal representation of an RMTIL's module structure is given in Figure 4. The MODIFIES and EFFECTS sets of a nonbasic template include those objects whose scope of existence is not limited to the scope of the template. For example, assume $T_{i,j}$, a template within T_i, contains the following segment of code:

allocate (*a*);
...
dispose (*a*);

or

open-file(afile);
read (afile, *b*);
close-file(afile);

a and *afile* do not appear in the EFFECTS or MODIFIES list of T_i.

Figure 5 depicts the RMTIL specification for the hypothetical module presented in Figure 2.

5. INTRACONNECTION OPERATIONS

This section is concerned with the operations incorporated into the RMTIL language to facilitate support for a programmer (user) who is building programs from reusable templates. Templates can be reused in different ways to modify or construct a module. The user may ask to add a template or substitute one known template for another, list a group of selected templates to be added/substituted, or look for a template to be substituted for a given template.

What follows is an outline of the possible ways in which intraconnection operations can be requested

IF : (c) $|\rightarrow$ (T_1 | T_2)
CASE : (c_1; c_2; ...; c_n) $|\rightarrow$ (T_1; T_2; ...; T_n | NEXT)
WHILE : (t_i-INVARIANT) $|\rightarrow$ (($T_{i.1}$, $T_{i.2}$, ..., $T_{i.n}$, t_i) | (NEXT))
UNTIL : (($T_{i.1}$, $T_{i.2}$, ..., $T_{i.n}$), (t_i-INVARIANT)) $|\rightarrow$ (($T_{i.1}$, $T_{i.2}$, ..., $T_{i.n}$, t_i) | (NEXT))

where

$\|\rightarrow$	: the correspondent-or operator
;	: condition separator
,	: sequence operator
t	: basic or composite template
T	: (t_1, t_2, ..., t_n), n > 0
NEXT	: next template in the control flow of the outermost template or a module

| and the $|\rightarrow$ operator is defined as follows

(c_1; c_2; ...; c_n) $|\rightarrow$ (T_1; T_2; ...; T_n | NEXT)
IF c_1 THEN T_1
ELSE IF c_2 THEN T_2
.
.
.
ELSE IF c_n THEN T_n
OTHERWISE NEXT

Figure 3. Control flow representations in RMTIL.

by a user:

1. Substitute template t_{old} by t_{new} in module m: Substitute (m, t_{old}, t_{new})
2. Add template t_{new} in position p of module m: Add (m, t_{new}, p)
3. Substitute the set of templates $\{t_{old}\}$ by $\{t_{new}\}$ in module m: Substituteall $(m, \{t_{old}\}, \{t_{new}\})$
4. Link the set of templates $\{t\}$ with the given control flow to create a module m: Link $(\{t\}, \text{CONTROL-FLOW}(m), m)$

The above operations assume that the substituting templates are known. In the cases where a template is not known, a search in SADB is required to find the desired template(s). Once the search is successfully carried out, one of the above operations will take place. More detail about these operations is provided in the next section.

Figure 6 illustrates a revised version of module "example" shown in Figure 2. In this example, template T2 is replaced by a new version of the template. Figure 7 illustrates the RMTIL code for the revised "example." Figure 8 is a more realistic illustration of Figure 7. In this figure, only the substituted template is coded. Nonmodified templates are referenced by their identifiers.

6. INTRACONNECTION PROCESS

The major tasks of the intraconnection process include checking for the overall consistency and integrity of the module that contains the template(s) involved and invoking other components of the ROPCO environment for required action. To perform the consistency check, ROPCO use an automatic verifier. The next subsection provides a short introduction to the module/template verification process used in the ROPCO environment.

6.1 Verification Process

The details of constructing proofs of program verification in the ROPCO environment are beyond the scope of this article. Verification of template addition/deletion/substitution is in general more complex than the verification of module interconnections but less cumbersome than overall program verification. Considering the fact that RMTIL as-

```
MODULESP (( NAME: xxx),
  ATTRIBUTE :
        (FUNCTION: yyy,
         LANGUAGE: zzz,
         ID-VERSION: ttt),
  VER-CONDITIONS :
        (REQUIRES :    (...), / module pre-conditions  /
         EFFECTS  :    (...), / module post-conditions /
         MODIFIES :    (...)),
  CONTROL-FLOW: (T1, T2, ..., Tn))
END xxx.

TEMPLATESP ((IN xxx),
   {template-id: (VERSION (x)),

        [INVARIANT:   (invariant for loops)],
        [DECREMENT: (decrement for loops)],
        REQUIRES:     (requirements list),
        MODIFIES:     (modification list),
        EFFECTS:      (effects on the template objects)
        [TEMP-CONTROL-FLOW: (list of template-ids [conditions])]
        })
END xxx.
```

Figure 4. Structure of an RMTIL module.

Note: [] denotes an optional item and {} denotes one or more repetitions.

```
MODULESP ((NAME : Example),

    ATTRIBUTE :(
      FUNCTION : hypothetical,
      LANGUAGE : mixed,
      ID-VERSION  : xxx-12-4),

    VER-CONDITIONS:(
      REQUIRES :(...), /list of imported objects, required resources,
                       and input arguments/
      EFFECTS  :(...), /list of export objects and outgoing
                       arguments/
      MODIFIES :(...)) /lists of all required objects modified by the
                       module/
    CONTROL-FLOW (T1, T2, T3, T4))

END Example.

TEMPLATESP ((IN Example),

    (T1:(VERSION (4),
         REQUIRES :(),
         MODIFIES :(i),
         EFFECTS  :(i := 0))),

    (T2:(VERSION (1),
         INVARIANT :(n > i),
         DECREMENT :(n - i),
         REQUIRES  :(i := 0 & n > 0),
         MODIFIES  :(i, cond),
         EFFECTS   :((i := n) | (i > n)),
         TEMP-CONTROL-FLOW((i < n) |→ (t2.1, t2.2, t2.3, T2) | NEXT)))

    (t2.1:(VERSION (1),
          REQUIRES :(),
          MODIFIES :(cond),
          EFFECTS  :())

    (t2.2:(VERSION (3),
          REQUIRES :(cond > 0),
          MODIFIES :(i),
          EFFECTS  :(cond = 1 |→ (i := i + 2)|(i := i + 1))))

    (t2.3 ...)

    (T3:(VERSION (2),
         REQUIRES :(),
         MODIFIES :(y),
         EFFECTS  :(y := x²prod + xb + c)))

    (T4:(VERSION (3),
         REQUIRES :(),
         MODIFIES :(y),
         EFFECTS  :(y => 0))))

END Example.
```

Figure 5. RMTIL specification of the module presented in Figure 2.

```
                MODULE Example {new version}

          BEGIN
T1              i := 0
T2              DO WHILE  i < n
                BEGIN
T2.1                  READ(cond,c)
T2.2                  IF cond =1 THEN
                            i := i - c
                      ELSE
                            i := i + c
                      END IF
T2.3                  PRINT (i)
                END WHILE
T3              y := prod * x
                y := (y + i) * x
                y := y + c
T4              IF y < 0  THEN
                            y := 1
          END {Example}
```

Figure 6. Revised version of module "example" in Figure 2.

signs the task of name-space consistency to other parts of ROPCO (PNET and PSBT), the required verifier needs only to check for the consistency of modules. Our approach to verification in the ROPCO environment is to compare the pre- and postconditions of the templates involved (or the new template) with those of the surrounding templates.

Unlike module interconnection, template intraconnection could have a local effect on other parts of a module. This local effect is not limited to name-space inconsistency; it might have an impact on the preconditions of the following template(s) as well. Therefore, the required verifier must be capable of performing the following tasks:

1. Determine the effect of the new template on the verification conditions of other templates.
2. Isolate and resolve the potential inconsistency of the affected template(s).
3. Approximate the impact of changing the affected template(s).

These tasks are somewhat analogous to the tasks of a sophisticated program verifier [25].

Since the intraconnections of the operations in RMTIL could be defined in terms of Add and Substitute, we will provide verification rules for these two cases in the subsections that follow. We use transformations to represent the Add and Substitute operations on a sequence of templates. Substitute is represented as follows:

$$T_1T_2 \ldots T_i \ldots T_n \Rightarrow T_1T_2 \ldots T_i \ldots T_n$$

Add is represented as follows:

$$T_1T_2 \ldots T_iT_{i+1} \ldots T_n \Rightarrow T_1T_2 \ldots T_iT_jT_{i+1} \ldots T_n$$

6.2 Substitute

This operation can be characterized in terms of the pre- and postconditions of the two templates involved. Based on the following definitions, two templates can be either functionally identical or functionally compatible with possible side effect.

Two templates are functionally identical

$$t_i \equiv t_j$$

if they can be used interchangeably without any unanticipated results.

Two templates t_i and t_j are functionally compatible

$$t_i \supset t_j$$

if t_j could be used in place of t_i and still provide the behavior required by the system.

A template and its associated verification conditions are denoted by

$$Pre_n\{t_n\}Post_n, Modify_n$$

where Pre_n is the set of minimum conditions required for execution of template n, $Post_n$ is the set of expectations from template n, and $Modify_n$ is the set of all objects whose states are modified in template n. $pre_n \subset pre_k$ indicates that pre_n is a subset of pre_k, which means that it does not assume more than pre_k. $post_n \supset post_k$ indicates that $post_n$ is a superset of $post_k$, which means that the new template guarantees all postrequirements.

The following propositions characterize the notions of being functionally identical or compatible for templates.

Proposition 1. Two templates t_{old} and t_{new} are functionally identical if and only if:

$$Pre_{new} \equiv Pre_{old}$$

$$Post_{new} \equiv Post_{old}$$

$$Modify_{new} \equiv Modify_{old}$$

Proof. Since the pre- and postconditions and the modify set are identical in both templates, all requirements of the succeeding templates will be fulfilled by the new template precisely the same way it was satisfied by the old template. Therefore, replacement of t_{old} with t_{new} will not affect the pre- and postconditions of the succeeding records. On

```
MODULESP ((NAME : Example),

    ATTRIBUTE :(
      FUNCTION : hypothetical,
      LANGUAGE : mixed,
      ID-VERSION  : xxx-12-5),

    VER-CONDITIONS:(
      REQUIRES :(...), /list of imported objects, required resources,
                       and input arguments/
      EFFECTS  :(...), /list of export objects and outgoing
                       arguments/
      MODIFIES :(...)) /lists of all required objects modified by the
                       module/
    CONTROL-FLOW (T1, T2, T3, T4))

END Example.

TEMPLATESP ((IN Example),

    (T1:(VERSION (4),
         REQUIRES :(),
         MODIFIES :(i),
         EFFECTS  :(i := 0))),

    (T2:(VERSION (2),
         INVARIANT :(n > i),
         DECREMENT :(n - i),
         REQUIRES  :(i := 0 & n > 0),
         MODIFIES  :(cond, c),
         EFFECTS   :((i := n) | (i > n),
         TEMP-CONTROL-FLOW((i < n) |→ (t2.1, t2.2, t2.3, T2) | NEXT)))

    (t2.1:(VERSION (2),
          REQUIRES :(),
          MODIFIES :(cond, c),
          EFFECTS  :())

    (t2.2:(VERSION (4),
          REQUIRES :(cond > 0),
          MODIFIES :(i),
          EFFECTS  :( cond = 1 |→ (i := i + c)|(i := i - c))))

    (t2.3 ...)

    (T3:(VERSION (2),
         REQUIRES :(),
         MODIFIES :(y),
         EFFECTS  :(y := x²prod + xb + c)))

    (T4:(VERSION (3),
         REQUIRES :(),
         MODIFIES :(y),
         EFFECTS  :(y => 0))))

END Example.
```

Figure 7. RMTIL specification of the module presented in Figure 6.

```
MODULESP ((NAME : Example),

    ATTRIBUTE :(
      FUNCTION : hypothetical,
      LANGUAGE : mixed,
      ID-VERSION  : xxx-12-5),

    VER-CONDITIONS:(
      REQUIRES :(...), /list of imported objects, required resources,
                       and input arguments/
      EFFECTS  :(...), /list of export objects and outgoing
                       arguments/
      MODIFIES :(...)) /lists of all required objects modifyied by the
                       module/
    CONTROL-FLOW (T1, T2, T3, T4))

END Example.

TEMPLATESP ((IN Example),

    (T1:( xxx-12-4:T1))

    (T2:(VERSION (2),
         INVARIANT :(n > i),
         DECREMENT :(n - i),
         REQUIRES  :(i := 0 & n > 0),
         MODIFIES  :(cond, c),
         EFFECTS   :((i := n) | (i > n),
         TEMP-CONTROL-FLOW((i < n) |→ (t2.1, t2.2, t2.3, T2) | NEXT)))

    (t2.1:(VERSION (2),
           REQUIRES :(),
           MODIFIES :(cond, c),
           EFFECTS  :())

    (t2.2:(VERSION (4),
           REQUIRES :(cond > 0),
           MODIFIES :(i),
           EFFECTS  :( cond = 1 |→ (i := i + c)|(i := i - c))))

    (t2.3:( xxx-12-4:t2.3))

    (T3:(xxx-12-4:T3))

    (T4:(xxx-12-4:T4))


END Example.
```

Figure 8. RMTIL specification of the module presented in Figure 6. Nonmodified templates are represented by their Id.

the other hand, if t_{new} and t_{old} are functionally identical, by definition, they can be interchanged without any unexpected results. Therefore, there must be no change in the modify set. The justification for adding Modify to the verification equation is that Modify could contain a set of modified objects in a template which may or may not be in the postcondition set. Ignoring members of this set may lead to overlooking the possibility of side effects on the subsequent templates. □

Proposition 2. Two templates t_{old} and t_{new} are functionally compatible, with possible side effect, if

$$Pre_{new} \subset Pre_{old}$$

$$Post_{new} \supset Post_{old}$$

Proof. Since $Pre_{new} \subset Pre_{old}$, by definition, pre_{old} will be satisfied whenever pre_{new} is satisfied. Also, since $Post_{new} \supset Post_{old}$, all postrequirements for $post_{new}$ are guaranteed. However, some results may

be produced that can affect the succeeding preconditions (side effect). If the behavior of a new template is the same as the old one, i.e.,

$$Modify_{new} \equiv Modify_{old}$$

there would be no further side effects; otherwise additional side effects are possible. □

The process of substitution of template *old* with a functionally compatible template, *new*, is similar to the process of Add, which is described in the next subsection.

6.3 Add

To add a template to a module, an automatic verifier can be used to compare the existing pre- and postconditions at the insertion position with the corresponding template conditions. The precondition for the insertion position could be found as follows.

If a template is being added at position p in the control flow of a program, then the precondition of position p, R_p, is determined by equation 1 below. This is analogous to Dijkstra's discussion of strongest preconditions [26]. The new template satisfies the insertion-point precondition if one of the following conditions exist:

$$Pre_p \equiv Pre_{new}$$

or

$$Pre_p \supset Pre_{new}$$

The postcondition $Post_p$ of the new template needs to be examined for possible side effects (the new templates may do more than the old one) on the preconditions of the subsequent templates.

$$R_p = \{\ldots\{\{R(t_1) \veebar \{Q(t_1) \cup MO(t_1)\}\} \veebar \{Q(t_2) \cup MO(t_2)\}\ldots\{Q(t_{p-1}) \cup MO(t_{p-1})\} \quad (1)$$

where R and Q denote pre- and postconditions, MO stands for the *Modify* set, $t_1, t_2, \ldots, t_{p-1}$ are the preceding templates, and the $\veebar$ operation, called **Precedence-OR**, is defined as follows. Let L, R and S be sets and m be a member which exist in both L and R denoted by m_l and m_r, after execution of

$$S := L \veebar R$$

S would contain m_r. For example, if

$$pre_i: \{a := 10, \quad b := 5, \quad c := 2\}$$

and

$$post_i: \{a := 20, \quad b := 5\}$$

then

$$pre_{i+1} := pre_i \veebar post_i$$

produces

$$pre_{i+1}: \{a := 20, \quad b := 5, \quad c := 2\}.$$

6.4 The Template-Substitute Algorithm

Figure 9 presents the Template-Substitute algorithm. This algorithm is designed to control the substitution of a template in a module. It first checks for name-space consistency between the candidate template and the module. If inconsistency exists, it calls the editor facility to remove the inconsistency, if possible. Once the template has been successfully checked for type/object consistency, the algorithm calls an automatic verifier to check for function compatibility between the present and candidate templates. Based on the verifier results, the algorithm proceeds by adding the template to the module and updating the relevant information. The last step is to call for compiling the template to generate the object code of the template.

6.5 Template Substitution when Substitutes are Not Known

Figure 10 depicts an outline for the Search & Substitute algorithm. This algorithm first calls **FindTemplate** to find one or more templates that satisfy the given specification based on the attribute list given. **FindTemplate** returns **templist**, which contains the set of pointers to the location of the template's code and its interconnection specification,[2] and the root of the VCT tree.

The templates are sorted based on their relevance to the given **attlist**. The algorithm selects one template at a time and calls the Template-Substitute algorithm. Based on the result of Template-Substitute, it either selects another template from the **templist** or terminates.

6.6 Substitution of Known Templates

The Substituteall and Link operations defined in section 5 could be described in terms of the Substitute and Add operations. Substituteall could be written as follows:

Algorithm Substituteall (m, set_{old}, set_{new})
BEGIN

[2] The location of the specification for each template is stored in the template node in the family tree in VCT of the SCM component of ROPCO.

Algorithm Template-Substitute (module, templ, presentempl) RETURNS subscond

```
Edit:            PROCEDURE {activates an interactive editor to modify templ
                 using modules PSBT and PNET, returns conditions of edit in
                 editcond}
Verify:          PROCEDURE {activates an automatic verifier to verify
                 newtempcond against oldtempcond and returns verification
                 conditions}
Add-PNET:        PROCEDURE {adds objects in the templ to the PNET}
Typecheck:       PROCEDURE {checks for name-space consistency between the
                 candidate template and the module, returns the type check
                 status in typecond}
inconsislist:    SET {set of inconsistent objects returned from Edit}
newtempcond:     POINTER {to the pre- and post-conditions of the candidate
                 template}
oldtempcond:     POINTER {to the pre- and post-conditions of the present
                 template}
module:          {module id}

 BEGIN

      REPEAT
            typecond  :=  Typecheck(templ,  module,  presentempl,
            inconsislist)

            IF typecond = INCONSISTENT
                  editcond := Edit(module, templ, inconsislist)
                  E-Flag := ON
            END IF

            IF {it is not possible to adapt the template}
                  subscond:= INCONSISTENT
                  EXIT
            END IF

      UNTIL typecond ≠ INCONSISTENT & editcond = SUITABLE

      IF E-flag = ON & editcond = SUITABLE
            modify  newtempcond if necessary
            subscond := Verify(newtempcond, oldtempcond)
      END IF

      IF subscond = COMPATIBLE
            Add-PNET(module, module-control-flow, templ)
            update PHDB

            IF E-flag = ON
                  Update PSBT
                  Update ADB
                  Update newtempcond
                  compile template and update VCT
            END IF
      END IF
  END {Template-Substitute}
```

Figure 9. Algorithm Template-Substitute.

```
WHILE {set_new} ≠ EMPTY
  t_old := Next(set_old)
  t_new := Next(set_new)
  Template-Substitute (m, t_new, t_old)
END WHILE
END {Substituteall}
```

and Link could be written as follows:

```
Algorithm Link (set_tem, CONTROL-FLOW, m)
BEGIN
  WHILE set_tem ≠ EMPTY
    t := Next(set_tem)
```

Algorithm Search&Substitute()

```
FindTemplate:      PROCEDURE {returns the set of templates in templist which satisfy the attlist}
position-template: PONITER {to the template to be substituted by the new one}
Next:              PROCEDURE {returns the next element in a list; if the list pointer points to the
                   header of the list, it returns the first element of the list}
attlist:           RECORD {specification of the desired template}
templist:          RECORD {pointers to the location of the template's code, its inter-connection
                   specification, and the root of the related VCT tree}

BEGIN
 Get(attlist)
 FindTemplate(attlist, templist)
 current := Next(templist)

 DO WHILE current ≠ NULL
       templ := templist(current)

       IF templ needs modification THEN
             editcond := Edit(module, templ, inconsislist)
       END IF

       subscond := Template-Substitute (module, templ, position-template)
       CASE subscode
             UNSOLVABLE : display "unsolvable case"
                   current := Next(templist)
             INCOMPATIBLE : current := Next(templist)
             COMPATIBLE :  current := NULL
       END CASE

 END WHILE
 END {Search&Substitute}
```

Figure 10. Algorithm Search & Substitute.

```
    p := Position(t, CONTROL-FLOW)
    Add (m, t, p)
  END WHILE
END {Link}
```

The Add algorithm is essentially the same as the Substitute algorithm with two exceptions. First, to add a template to a module, all names in the templates that would not cause type inconsistency in the module name space need to be copied from the declaration template (template 0) in the source module into the target module. Second, the verification process for addition of a module is different from that for substitution.

6.7 The Template-Addition Algorithm

The template addition algorithm is basically the same as the Template-Substitute algorithm with two differences. First, the third input parameter of the Add algorithm is the position of the insertion. Second, the verifier checks the verification conditions of the insertion point against the new template. All other steps are exactly the same as the Template-Substitute algorithm.

7. SUMMARY

The objective of this article was to present the design of an interconnection language for the ROPCO environment to facilitate the selection, customization, and interconnection of reusable components.

An interconnection language is a machine-processable specification language which provides the means for a system designer to represent the system in a concise, precise, and verifiable form. Once the system structure has been designed, it can be coded

in an interconnection language. This code is used to verify the design completeness and detect inconsistencies before the system is actually linked together.

Automatic processing of such formal specifications requires system-integrity and intermodule-compatibility verification. Conventional MILs are not concerned with the specification information (function of the system) or the detailed design information (how the modules implement their functions); they are concerned with architectural design information (how the modules fit together).

Conventional MIL models were examined, their strength and weakness were discussed briefly, and a new type of model for interconnection of modules/templates was introduced. In the ROPCO environment, it is not possible to use a conventional MIL. Rather, a model and language is needed that provide capabilities to facilitate intraconnecting (i.e., substituting a template in a module, assembling a number of templates within a module, or creating a module) as well as interconnecting modules. This new language, which is based on CLU, is called RMTIL.

The RMTIL module provides four specifications for each software module which it represents: attribute specification, module structure, module interconnection specification, and template specification. Template specification consists of template pre- and postconditions, other verification conditions, and the structure of the template (template control flow) if the template is composite.

RMTIL is part of the ROPCO environment and is specifically designed to use and interoperate with the other components of ROPCO. RMTIL consolidates other parts of ROPCO and uses the SCM structures, i.e., VCT, PSBT, PNET, and PHDB. Type checking for name-space consistency is not one of the tasks of the automatic verifier in the ROPCO environment; RMTIL uses PSBT and PNET to perform this task.

The ROPCO environment is under implementation and testing. The VCT, identification mechanism, PNET, PSBT, and tools for reverse engineering are being implemented in the Department of Mathematics and Computer Science of the University of Nebraska at Omaha and the Computer Science Department of Oklahoma State University [15, 27, M. K. Zand and D. Hiesterkamp, unpublished data].

ACKNOWLEDGMENTS

We thank the reviewers for their constructive comments on earlier drafts of this article.

REFERENCES

1. R. Prieto-Diaz and J. M. Neighbors, Module Interconnection Languages, *J. Syst. Software* 6, 307–334 (1987).
2. W. Tichy and M. C. Baker, Smart recompilation, in *Conference Record of the 12th Annual Symposium on Principles of Programming Languages*, AMC, New York, 1985, pp. 236–244.
3. D. E. Perry, Version control in the Inscape environment, in *Proceedings of the 9th International Conference on Software Engineering*, 1987, Monterey, CA, IEEE Computer Society, pp. 142–149.
4. D. E. Perry, Software interconnection models, in *Proceedings of the 9th International Conference on Software Engineering*, Monterey, CA, IEEE Computer Society, 1987, pp. 61–69.
5. J. A. Goguen, Reusing and Interconnecting Software Components, *IEEE Comp.* 19, 16–28 (1986).
6. D. E. Perry, The constructive use of module interface specifications, in *Proceedings of the 3rd International Workshop on Software Specification and Design*, London, IEEE Computer Society, 1985, pp. 179–180.
7. D. E. Perry, The Inscape Program Construction and Evolution Environment, Technical Report, Computer Technology Research Laboratory, Murray Hill, NJ, AT&T Bell Labs, 1986.
8. W. F. Tichy, Software Development Control Based on System Structure Description, Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1980.
9. N. Hebrman, et al., *A Compendium of Gandalf Documentation*, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1981.
10. W. Tichey, Smart Compilation, *ACM Trans. Progr. Lang. Syst.* 8, 273–291 (1986).
11. S. S. Yau and J. J. Tsai, Knowledge Representation of Software Component Inter-Connection for Large-Scale Software Modifications, *IEEE Trans. Software Eng.* SE-13, 355–361 (1987).
12. D. M. Ritchie, *The C Programming Language, Reference Manual*, Murray Hill, NJ, AT&T Bell Labs, 1980.
13. B. W. Lamson and E. E. Schmidt, Organizing Software in a Distributed Environment, *ACM SIGPLAN Not.* 18 (1983).
14. D. B. Leblang and G. D. Mclean, Computer-Aided Software Engineering in a Distributed Workstation Environment, *SIGPLAN Not.* 19, 104–112 (1984).
15. M. K. Zand, M. H. Samadzadeh, and K. M. George, ROPCO—An environment for micro-incremental reuse, in *Proceedings of IEEE International Phoenix Conference on Computers and Communications*, Omaha, NE, pp. 347–355.
16. W. Teitelman and L. Mainster, The Interlisp Programming Environment, *Computer* 14, 637–654 (1985).
17. G. E. Kaiser and A. N. Habermann, An environment for system version control, in *CompCon '83*, IEEE Computer Society Press, 1983, pp. 415–420.

18. W. J. Tracz, Software Reuse: Motivations and Inhibitors, *Proceedings of COMPCON87*, 1987, Washington, DC, IEEE Computer Society, pp. 358–368.
19. C. Withrow, Error Density and Size in Ada Software, *IEEE Software* (1992).
20. M. K. Zand, M. H. Samadzadeh, and H. Saiedian, Version Management for ROPCO–A Micro Incremental Reuse Environment, Technical Report UNO-CS-TR-92-4, *Math and Computer Science*, University of Nebraska at Omaha, Omaha, Nebraska, 1992.
21. M. K. Zand, ROPCO—An Environment for Micro-Incremental Reuse, Ph.D. Thesis, Computer Science Dept., Oklahoma State University, Stillwater, Oklahoma, 1990.
22. M. K. Zand, M. H. Samadzadeh, and K. M. George, Minimizing ripple recompilation in a persistent software environment, in *Proceedings of the ACM Computer Science Conference*, 1990, pp. 166–172.
23. B. Liskov and J. Guttag, *Abstraction and Specification in Program Development*, The MIT Press, Cambridge, Massachusetts, 1988.
24. B. Liskov, et al., Abstraction Mechanisms in CLU, *Commun. ACM* 564–574 (1977).
25. M. S. Moriconi, A Designer/Verifier's Assistant, *IEEE Trans. Software Eng.* 387–401 (1979).
26. E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
27. L. E. Swanson and M. H. Samadzadeh, A reusable software catalog interface, in *Proceedings of the SAC'92*, ACM, pp. 1076–1083.