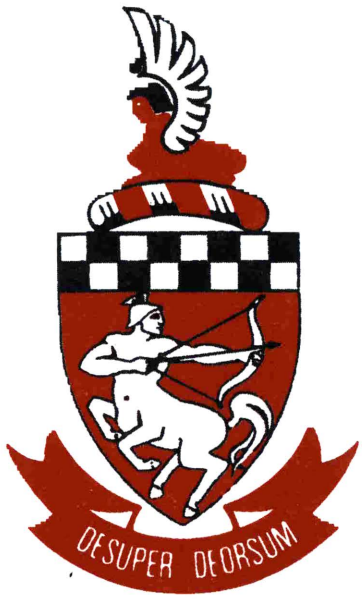
South African Computer Journal

Number 25
August 2000

Suid-Afrikaanse Rekenaar-Tydskrif

Nommer 25
Augustus 2000



The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarvereniging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

World-Wide Web: `http://www.cs.up.ac.za/sacj/`

Subscriptions

	Annual	Single copy
Southern Africa	R80.00	R40.00
Elsewhere	US$40.00	US$20.00

An additional US$15 per year is charged for airmail outside Southern Africa

to be sent to:

Computer Society of South Africa
Box 1714, Halfway House, 1685
Phone: +27 (11) 315-1319 Fax: +27 (11) 315-2276

In Memoriam:
Stef Postma

South Africa has lost one of her most colourful and eminent computer scientists. Professor Stef Postma passed away peacefully in his sleep on May 5, 2000 after a short illness. He will be remembered for his forthright views and total integrity. Never a man to shy from controversy, he always debated his position with vigour, displaying his extensive vocabulary at every opportunity.

Those who knew him mourn the loss of a very good friend.

Stef was born on August 10, 1938 in Graaff-Reinet and matriculated from Höerskool Linden in Johannesburg. He majored in geology and mathematics at the University of the Witwatersrand and graduated with honours in mathematics from that university. Stef devoted much of his life to promoting computer science as a science and to this end spent a lot of energy and time defining syllabi for undergraduate and honours courses at our universities. He was the prime mover in creating the South African Institute of Computer Scientists and Information Technologists (SAICSIT) in 1982, providing a professional body to represent the interests of local computer scientists. He was also instrumental in establishing Quaestiones Informaticae (now the South African Computer Journal) which afforded South African computer scientists the opportunity to publish papers locally in a refereed journal.

-Doug Laing

Links2Go "Computer Science Journals" Award

The SACJ production editing team received the following unsolicited e-mail:

The page at <http://www.cs.up.ac.za/sacj/>, was selected as a Links2Go "Key Resource" in the Computer Science Journals topic, at <http://www.links2go.com/topic/Computer_Science_Journals>.

How your page was selected:

Each quarter, Links2Go samples millions of web pages to determine which pages are most heavily cited by web pages authors, such as yourself. The most popular pages are downloaded and automatically categorized by topic. At most 50 of the pages related to a topic are selected as "Key Resources." Out of 50 pages selected as Key Resources for the Computer Science Journals topic, your page ranked 19th. For topics like Music, where there are a large number of interested authors and related pages, it is harder to achieve selection as a Key Resource than for a special-interest topic, such as Quantum Physics.

The Links2Go Key Resource award differs from other awards in two important ways. First, it is objective. Most awards rely on hand selection by one or more "experts," many of whom have only looked at tens or hundreds of thousands of pages in bestowing their awards. Selection for these awards means no more than that one person, somewhere, noticed your page and liked it enough to select it. The Key Resource award, on the other hand, is based on an analysis of millions of web pages. Any group or organization who conducts a similar analysis will arrive at similar conclusions. When Links2Go says your page is a Key Resource, we mean that your page is one of the most relevant pages related to a particular topic on the web today, using an objective statistical measure applied to an extremely large data set.

Second, the Key Resource award is exclusive. We get literally hundreds of people requesting that their page be added to one or more topics per week. All of these requests are denied. The only way to get listed as a Key Resource is to achieve enough popularity for our analysis to select your pages automatically. We do not accept fees, offers of link exchanges, free advertising, or bartered livestock as inducements to add new sites to our lists. Fewer than one page in one thousand will ever be selected as a Key Resource.

Once again, congratulations on your award! Links2Go Awards awards@links2go.com

A New Approach for Program Integration

Z-E Bouras[a] T Khammaci[b] S Ghoul[c]

[a]*Computer Science Department, University of Annaba, BP 12 Annaba, 23000, Algeria*
[b]*IRIN-University of Nantes, 2, rue de la Houssinire, 44321 Nantes Cedex 03, France*
[c]*Computer Science Department, Science College, Philadelphia University, PO Box 1101, Sweilah, Amman, Jordan*
[a]bourasz@yahoo.com, [b]khammaci@irin.univ-nantes.fr, [c]ghoul_said@yahoo.com

Abstract

Program integration combines independent enhancements to some version of a software system into a new system that includes the enhancements and the old system. Current approaches use the slice concept. The latter is a method for automatically decomposing programs by analysing their data and control flow. In this article, we show that the use of this concept in program integration is inadequate. We propose a new approach based on role concept which overcomes slice problems. A role models the behaviour by which a variable or a procedure intervenes in a given program.
Keywords: *Software Maintenance, Program Dependence Graph, Program Integration, Slice, Role*
Computing Review Categories: *D.2.2, D.2.3, D.2.6, D.2.6, D.2.9, D.3.4, E.1*

1 Introduction

Practical software systems are constantly changing in response to changes in user needs and the operating environment. Such changes often have to be developed concurrently and then combined in order to produce a new version.

Several researchers describe methods for effectively combining peoples' work in updating a software system. We focus on results from software engineering and programming languages that contribute to a closely related aspect of software maintenance: program integration.

Program integration combines various independent enhancements of some version of a software system into a new system that includes the semantics of both the enhancements and the old system. Program integration for imperative programs is of interest because many practical systems are written in that style [3]. Horwitz *et al* introduce the idea of merging changes to while-programs using the slicing concept [11]. A program slice is a subset of a program whose execution trace is independent of the rest of the program [16]. Informally, a slice provides the answer to the question "What program statements potentially affect the value of variable v at statement s?".

Slicing has the advantages of simplicity, safety and efficiency [3]. However, we cannot obtain abstraction levels and there are as many slices as instructions, in a given program. This means that we cannot reason at an abstraction level higher than the individual instructions. As a result, this approach is inadequate for practical program integration [6]. We propose a new approach which overcomes these problems. It is based on the concept of a role [2, 7]. A role allows us to raise the abstraction levels of reasoning and it is not dependent on the number of instructions but rather on the number of different variables and procedures in a program. Its application to program integration would be more suitable than slicing that is advocated by current researchers.

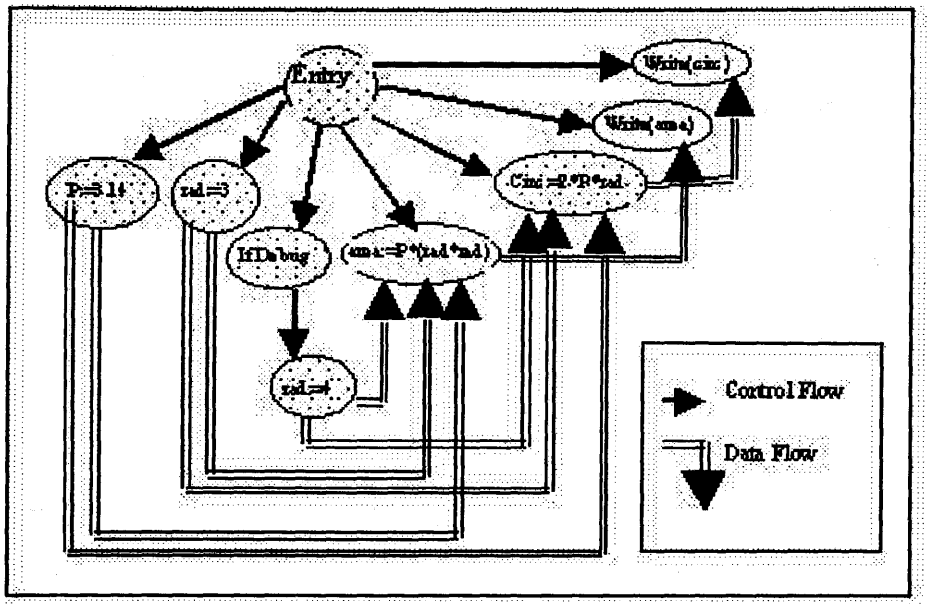
In the following section, we present the current approaches to program integration. In section 3, we present our formalism for modeling a given program. From this, and in section 4, we show the process by which we decompose program into roles. Section 5 presents our approach to program integration and ATLACY, a prototype system for integrating programs.

2 Current Approaches

Many program integration approaches have been proposed [11, 17, 4]. Changes in the behaviour of a given program are detected and preserved in the integrated program by using the concept of a slice.

Program slicing is a supporting technology for program integration. A program slice is a subset of a program whose execution trace is independent of the rest of the program. Weiser [16] has developed the idea of program slicing for debugging and parallel execution. It consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a slicing criterion.

We can use behavioural projections to decompose a programs semantics into independent parts. A behavioural projection is the part of the behaviour of a program visible from a particular vantage point, such as an output variable or output stream. Such projections help us understand complex systems. Program slicing is a way to materialize behavioural projection. The move of the slice to a new context will be valid, provided that the behavioural projection is appropriate in the new context.



a. Program Dependence Graph

```
Program Base
P := 3.14
rad := 3
if Debug then
   rad :=4 fi
area := P*(rad *rad)
circ := 2 * P*  rad
Write (area)
Write (circ)
End.
```

b. Program Base

Figure 1: Program Dependence Graph (PDG) of Base

From this we can see program integration as a sequence of steps; (1) decompose the program into disjoint behavioural projections, (2) replace some projections with new versions and (3) recombine the new versions with the unaffected components.

To integrate the program called Base shown in figure 1 and with its variants, current approaches start by constructing a Program Dependencies Graph or PDG. A PDG is a directed graph explicitly representing program dependencies [8]. The vertices represent the instructions of a program and the edges, control and data dependencies. Each PDG is decomposed into slices. For a vertex v of a program dependence graph G, the slice of G with respect to v is a graph containing all vertices on which v has a transitive flow or control dependence [4, 15]. Thus, the vertices of a slice are all vertices that can reach v via flow and/or control edges. In figure 1, we show the PDG (figure 1.a) of program Base (figure 1.b). We have shaded all vertices corresponding to the slice of circ:= 2 *P *rad.

By using extracted slices a set of changed variant instructions with regard to Base are identified as well as the set of preserved instructions. Two or more instructions are equivalent if their slices are equivalent [11, 13, 17]. From these sets, they construct a PDG, called GM, corresponding to the PDG of the integrated program. Finally, they check if there is interference. If not, an integrated program is generated from the GM graph. There are two possible ways in which the graph GM can fail to represent a satisfactorily integrated program. They are referred to as "Type I interference" and "Type II interference." Type I interference criterion is based on a comparison of slices of the variant graphs and with those of GM. If the slices, according to the changed entities in a given variant are different in variant graphs and GM then we get "Type I interference". "Type II interference" appears in the attempted reconstitution of merged program from GM. It is possible that reconstitution is not feasible because of the organization of instructions [17].

Binkley *et al* propose an improvement of this approach while applying it to procedural programs [4]. A semantic justification for the program integration algorithm of Horwitz and an alternative formulation of this approach based on Brouwerian algebra are presented in [13]. The algebraic laws that hold in such algebras are used to restate the algorithm and to prove properties such as associativity of consecutive integrations.

However, to proceed with changes, it is necessary to understand the programs. The latter is much easier when we have higher abstraction levels [10, 12]. The fact that one or several instructions can be shared by different slices makes it impossible to obtain abstraction levels. Also, in a given program, the number of slices is equal to the number of instructions. This makes the slice approach inadequate for integrating large programs.

We propose a structured representation which overcomes these problems. It is based on the concept of role.

A role models the behaviour which a variable or procedure displays in a program [7]. As in current approaches, we integrate programs written in a restricted imperative language. It is restricted by the fact that it uses only reading, assignment, alternate, loop, writing, procedure declaration and call instructions.

In our approach, an imperative program is modeled formally by an internal form that makes flow dependencies explicit. From this internal form we automatically decompose programs in roles and role systems. Roles are compared and merged in order to produce a new version of program.

3 Program Modeling

To permit automatic data and control analysis (dependence relationships) which are implicit in the program, their explicit representation in internal format is needed.

3.1 Internal form

For the time being, we are interested in a restricted imperative programming language whose constructions are reading (Read), assignment (:=), alternate (If), loop (While), writing (Write), call (Call) and procedure declaration (Procedure). In the context of program comprehension and program modification, a dependence relationship is defined formally by the triplet: <Ac, Ctr, As>.

It means that target actions Ac and source actions As have a dependence relationship according to the constraint Ctr [9, 5]. Elementary actions Ac and As are formally defined by the cartesian product: Var × Act × IdDep, where Var can be a program, control, call variable or a formal parameter. Act can be a definition, control or reference action. IdDep is a unique identifier corresponding to an instruction site, an effective parameter or to a formal parameter. Ctr is defined by the triplet: <Cdt, Exp, Sem>

Cdt is a condition which must be true to execute actions Ac. It expresses the control constraint. Exp is an expression to evaluate actions Ac. It expresses the data constraint. The dependence relationship Sem can be a flow dependence (FD), control dependence (CD), flow and control dependence (FCD), formal parameter flow (FPF), actual parameter flow (APF).

The internal form of a given program consists of modeling data and control structures.

3.2 Modeling Data Structure

Our data structures are composed of variables with the following types: integer, real, boolean, character. Actions on data structure are declaration (Declare), assignment (Assign), reading (Read), writing (Write) and interface actions (In, Out).

For each declaration or assignment instruction of a variable we associate one or more dependencies. An assignment instruction "X:= B op ... Y;" of a variable X defined in a procedure P needs an assignment action (Assign) to variable X which depends on one or more interface actions (In, Out) of the implied variables in the right hand side of the assignment instruction (B, ..., Y).

3.3 Modeling Control Structure

Control structures are composed of procedure declarations, calls, alternates and loop instructions. The main program is considered as a particular procedure. In addition to In and Out actions, the actions on control structure are test (If), loop (While), procedure declaration (Enter), and Call.

Procedure declaration instructions are modeled by a dependence between an Enter action and interface actions associated with formal parameters of this procedure.

Call instructions are modeled by a dependence between a call action and one or more interface actions associated with actual parameters of this procedure.

Alternate and loop instructions. We associate respectively with each alternate or loop instruction an alternate (if) or loop (while) action. All instructions defined in the body depend on the alternate or loop action.

Every program is seen as a set of triplets. Being formal, this modeling allows automatic role decomposition.

4 Program Decomposition

The decomposition that we propose is due to the logical structure of program and programming concepts. An imperative program is seen as a role system where a role models the behaviour with which a variable or procedure intervenes in this program.

4.1 Concept of Role

Program decomposition takes a program and makes objects and relationships out of it. The objects and relationships facilitate analysis transformation and extraction of further information [1]. From a program in internal form we automatically extract objects, in our case roles, and their systems. This role system represents the logical structure of program. It has a scheme composed of its name, set of roles and set of relationships between roles.

4.1.1 Extraction of Information Role

An information role models program variables. It has the same name and represents its data structure. Behaviour extraction of information role X consists of identifying the triplets having the following form:

$$\{<X, Act1, idep1>, Ctr, <Y, act2, idep2> / Act1 = \{Declare, Assign, Read, Write\}\}$$

4.1.2 Extraction of Control Role

Every procedure is modeled by a control role. It has the same name as the procedure and represents its control

```
program Base;
var area : real ;
    Pi : real ;
    radius : real ;
   Circ : real ;
   debug : integer ;
procedure prod( x,y,z : real ; varu : real ) ;
var t : real ;
begin read ( t ) ;
if ( T = 0.0 ) then u := x*y*z
else u := x*y*z/t ;
end ;
begin Pi := 3.14 ;
Radius = 3.0 ;
read ( debug ) ;
if ( Debug = 1 ) then Radius := 4.0 ;
prod( Pi, Radius, Radius, Area ) ;
write ( Area ) ;
Prod( 2.0, Pi, Radius, Circ ) ;
write ( circ ) ;
end .
```

a. Program Base

```
program A ;
var area : real ;
    Pi : real ;
    radius : real ;
    Circ : real ;
    debug : integer ;
    Diam : Real ;
procedure prod( x,y,z : real ; varu : real ) ;
var t : real ;
begin
read ( t ) ;
if ( T = 0.0 ) then u := x*y*z
else u := x*y*z/t ;
end ;
begin
Pi := 3.14 ;
Radius := 3.0 ;
Diam := 2.0*Pi ;
read ( debug ) ;
if ( Debug = 1 ) then Radius := 4.0 ;
prod( Pi, Radius, Radius, Area ) ;
write ( Area ) ;
Prod( 1.0,Diam, Radius, Circ ) ;
write ( circ ) ;
end .
```

b. Variant A

```
program B ;
var area : real ;
    Pi : real ;
    Rad : real ;
    Circ : real ;
    debug : integer ;
procedure prod( x,y,z : real ; varu : real ) ;
var t : real ;
begin
read ( t ) ;
if ( T = 0.0 ) then u := x*y*z
else u := x*y*z/t ;
end ;
begin
Pi := 3.1416 ;
Rad := 3.0 ;
read ( debug ) ;
if ( Debug = 1 ) then Rad := 4.0 ;
prod( Pi, Rad, Rad, Area ) ;
write ( Area ) ;
Prod( 2.0, Pi, Rad , Circ ) ;
write ( circ ) ;
end .
```

c. Variant B

Figure 2: Program Base and its variants

structure. The extraction of the behaviour of a control role X consists of identifying the triplets having the following form:

$$\{<X,Act3,idep3>,Ctr,<Y,act4,idep4> / Act3 = \{Enter,Call,If,While\}\}$$

The procedure concept allows software decomposition which is appropriate for the optimization of the software's implementation. However, different calls to the same procedure (P1) involve two problems for program comprehension and program modification. The first difficulty is about the information roles that may be modified by the procedure P1. The second one is related to the dependencies between involved roles.

In a procedure P1 which is called from many program points, a single instruction that modifies a formal parameter does not belong to the behaviour of just one variable but to the behaviour of all corresponding actual parameters. This is in contradiction with our fragmentation where each program instruction should belong to just one role. In order to resolve this problem, we substitute the formal parameters in the body of P1 with the actual parameters. In this way, each modification instruction will not belong to the formal parameter but to the role associated to the actual parameter.

For example, in the program Base (figure 2.a), the modification instruction "U:= X*Y*Z;" of formal parameter U of procedure Prod belongs to the behaviour of Area and to the behaviour of Circ involved respectively in the call instructions "[24] Prod(PI, Radius, Radius, Area);" and "[26] Prod(2, PI, Radius, Circ);" (figure 3.b). The substitution of formal parameters by actual parameters of procedure Prod in this instruction gives, respectively, instruction "[14] Area:= PI*Radius*Radius;" and instruction "[14] Circ:= 2*PI*Radius;". Thus, the first instruction belongs to role Area and the second to role Circ.

The second problem with the procedure concept concerns the formal parameters which hide the behaviours of user variables, making the program understanding and modification very difficult. In our approach we do not represent the formal parameters in the role system but we use them in order to identify the dependencies between actual parameters of each procedure call. In particular, we compute the summary relationships which are dependencies between input actual parameters and output actual parameters by making abstractions of the body of the procedure [14].

For example, in the call instruction "[24] Prod(PI, Radius, Radius, Area);" the role Area depends on the data and the control flow of roles PI and Radius because in the body of Prod the formal parameter U depends on data and control flows of formal parameters X, Y and Z.

4.1.3 Extraction of Role System

Extraction of role system S consists of identifying the pairs <X, Y> which means that the role X depends on the role Y. This extraction needs the identification of the following set:

$$E = \{<X, Action, Idep>, Ctr, <Y, Action, Idep> / X \neq Y\}$$

4.2 Role Scheme

In the context of program comprehension and program modification, we have identified for every role a scheme describing its type, behaviour and its interface.

Role Type. Two types are identified: control and information roles. The former is about the control structure and the latter about data structure. A particular information role, named Screen, is added to represent the written instructions.

Role behaviour. The actions defining and modifying a role give its behaviour. We have identified four actions on information roles: Declare, Assign, Read and Write. Actions on control roles are: Enter, Call, If and While. To permit an encapsulation of actions, a composed interfacing of entrance (In) and exit actions (Out) was provided to every role.

Role interface. A roles has interactions with other roles through its interface. To facilitate the behaviour comprehension of an information role, we display the instructions that define and modify the corresponding entity rather than its actions. In the case of a control role, we display the alternate and iteration instructions included in the body of a procedure, followed by every call to the procedure. In the case of a recursive procedure, call instructions are in the body of procedure. For example, from the program Base in figure 2.a, we can extract an information role scheme of Radius (figure 3.a), and a control role scheme of Prod (figure 3.b).

4.3 Role System Scheme

A role system has a scheme describing its name, information roles, control roles and a set of dependence relationships. In figure 3.c, we present the scheme of the role system of Base. For example <Area, Radius> means that the role Area depends on data flow from the role Radius.

5 Program Integration

From program decomposition we can integrate programs. The integration process consists of classifying roles on specific sets, merging results in order to produce a role system of the new version and generating the source code of a new version from this.

We propose to apply our approach to integrate programs Base, A and B of figure 2. Program A (figure 2.b) includes supplementary instructions to calculate diam. B (figure 2.c) includes a change of instructions (PI:= 3.1416 instead of PI:= 3.14) and a change of name (rad instead radius). The role decomposition step gives the following roles for each program respectively:

Base: Area, Base, Circ, Debug, PI, Prod, Radius, Screen, T.

A: A, Area, Circ, Debug, Diam, PI, Prod, Radius, Screen, T.

B: Area, B, Circ, Debug, PI, Prod, Rad, Screen, T.

Changes can be caused by a local change according to a given role (intra-role) or by a global change between roles (inter-roles). Changes may be semantic in nature or not. Semantic changes may lead to unforeseen interference that leads to failure of the integration process. Our program integration is based on intra-role and inter-role comparisons.

5.1 Intra-Role Identification

Intra-role changes concern the change of name or behaviour of a given role, without modification of its interface. It is then necessary to sort roles according to types of changes to identify semantic changes. By using role systems and roles worked out in the first, we establish four sets of roles: UR, CRN, CRB and CRNB.

Set UR contains Unaltered Roles in all programs. In this case the resulting role is the role of Base. In our example, we have in this set: Area, Circ, Debug, Prod, Screen and T.

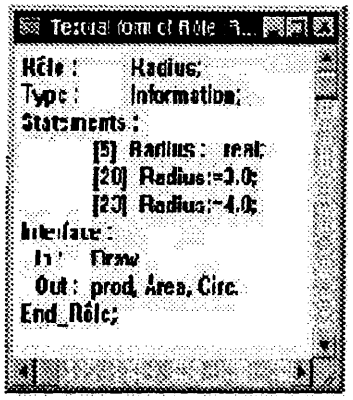
CRN concerns the set of Roles having Changed Names but the same behaviour. Informally, roles of CRN are the roles having only syntactic changes of their names in one or several variants. In our example, Base (changed in A and B) and Radius (Rad in variant A) belong to this set. The resulting role has as name a concatenation of variant name roles.

CRB concerns Roles having Changed Behaviours but keeping the same name. Roles of CRB have semantic changes in one or several instructions in the variants. In our example, role PI has the same name in all programs. The change of PI:=3.14 to PI:=3.1416 implies that this role belongs to CRB. If role behaviour has changed in a variant then the variant's role will be the resulting role. Otherwise we have interference.

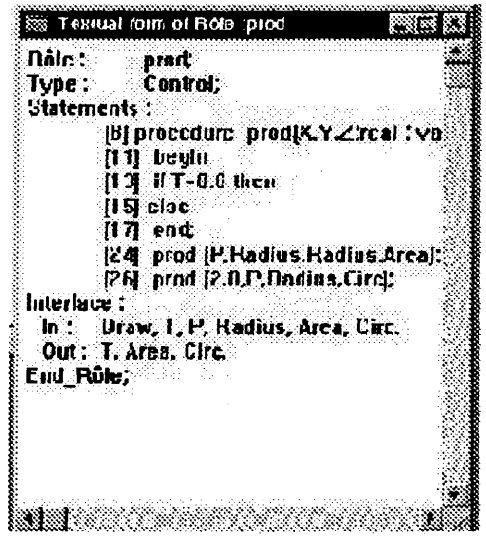
Informally, CRNB roles involve both semantic and syntactic changes in the variants. CRNB represents the deleted roles from Base or the inserted roles in the variants. In this case, if role belongs only to Base then it is eliminated during the integration. Then it will be eliminated in the new version. If it belongs only to variants, it is interpreted by its insertion in one or several variants then inserted into the new version. Otherwise we have interference.

5.2 Inter-role Identification

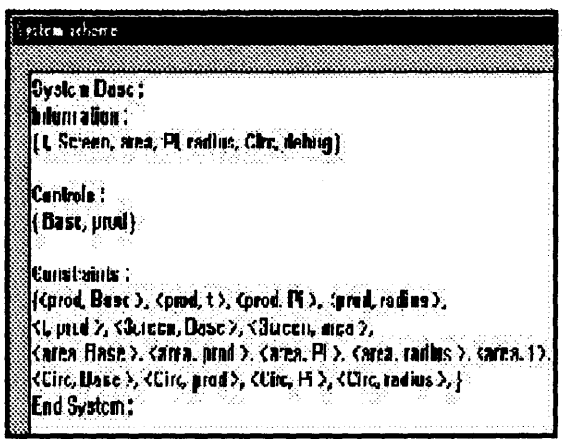
Inter roles changes concern the changes of global dependencies between roles. To identify global changes we de-



(a) Information role radius



(b) Control role Prod



(c) Role System of Base

Figure 3: Information, Control Roles and Role System schemes

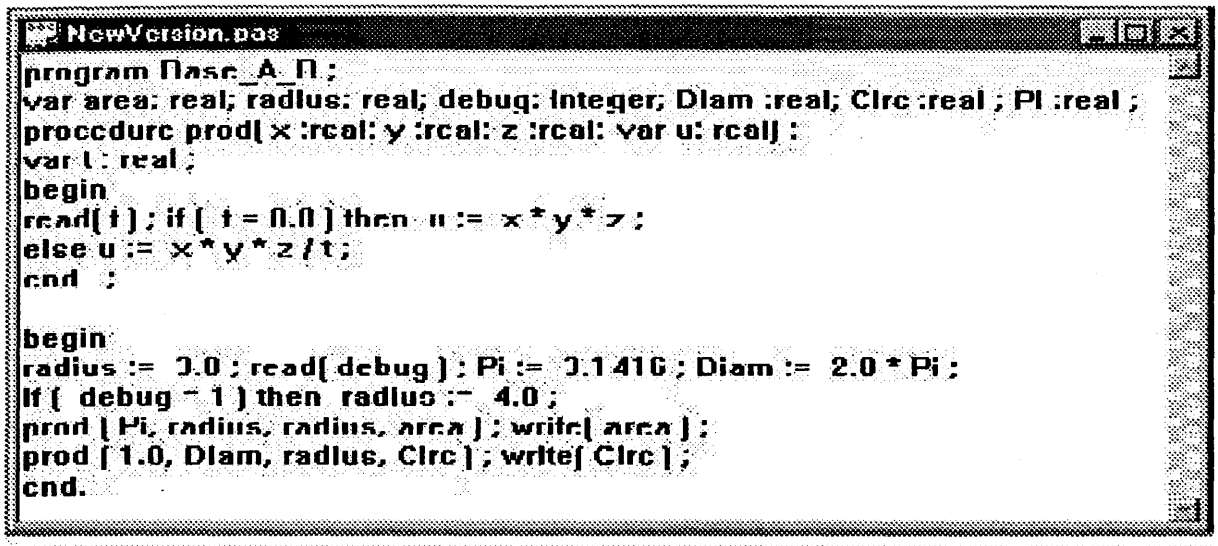


Figure 4: Source Code of the new version

fine four sets of relation: UD, SDC, TDC, STDC.

UD (Unaltered Dependencies) concerns dependencies having identical source and target roles. In this case, we insert dependencies of Base in the role system of new version. SDC (Source Dependencies Changed) is the set of dependencies where targets are the same and sources are different. TDC (Target Dependencies Changed) is the set of relations where sources are the same and targets are different. Finally STDC (Source and Target Dependencies Changed) concerns the set of relations where the source and targets are different. For every element of SDC, TDC or STDC the resulting relations are dependencies of variants. Finally we generate the source code of the resulting program.

5.3 Generating a New Version

We generate the integrated version from the role system and roles selected previously. The order of instructions is done according to the constraint Ctr (defined in section 3.1). Indeed Ctr is composed by a triplet:
< Cdt, Exp, Sem >.
According to the Cdt values and Exp we can order the internal form of the program and write its source code. We show in figure 4 the source code of new version of our application.

5.4 ATLACY: An Automated Tool for evoLving legACY code

We have designed and implemented a prototype based on our approach named ATLACY. It has been developed in a Visual C++ environment and is formed by five modules: Model, Decompose, Sort, Select and Generate (figure 5).

In the first step, Model carries out lexical, syntactic and semantic analysis of a given PASCAL source code program. It gives a table of variable definitions and references and a table of call points. From these tables, it elaborates program in internal form and instrumented source code. Then, Decompose extracts role and role system information of every program and displays to the user the list of involved roles and role systems. Sort deterrence the sets: UR, CRN, CRB and CRNB of roles and the sets: UD, SDC, TDC, STDC of relationships. After this step, and if there is no interference, Select chooses the roles and role system corresponding to the new version. Finally, Generate reconstitutes the source code of new version.

6 Conclusion

The current approaches of program integration are based on the concept of slices. Slices are fragments of programs that can share instructions. This does not permit one to obtain abstraction levels required for the understanding and therefore modifying the program. Besides, there are in a

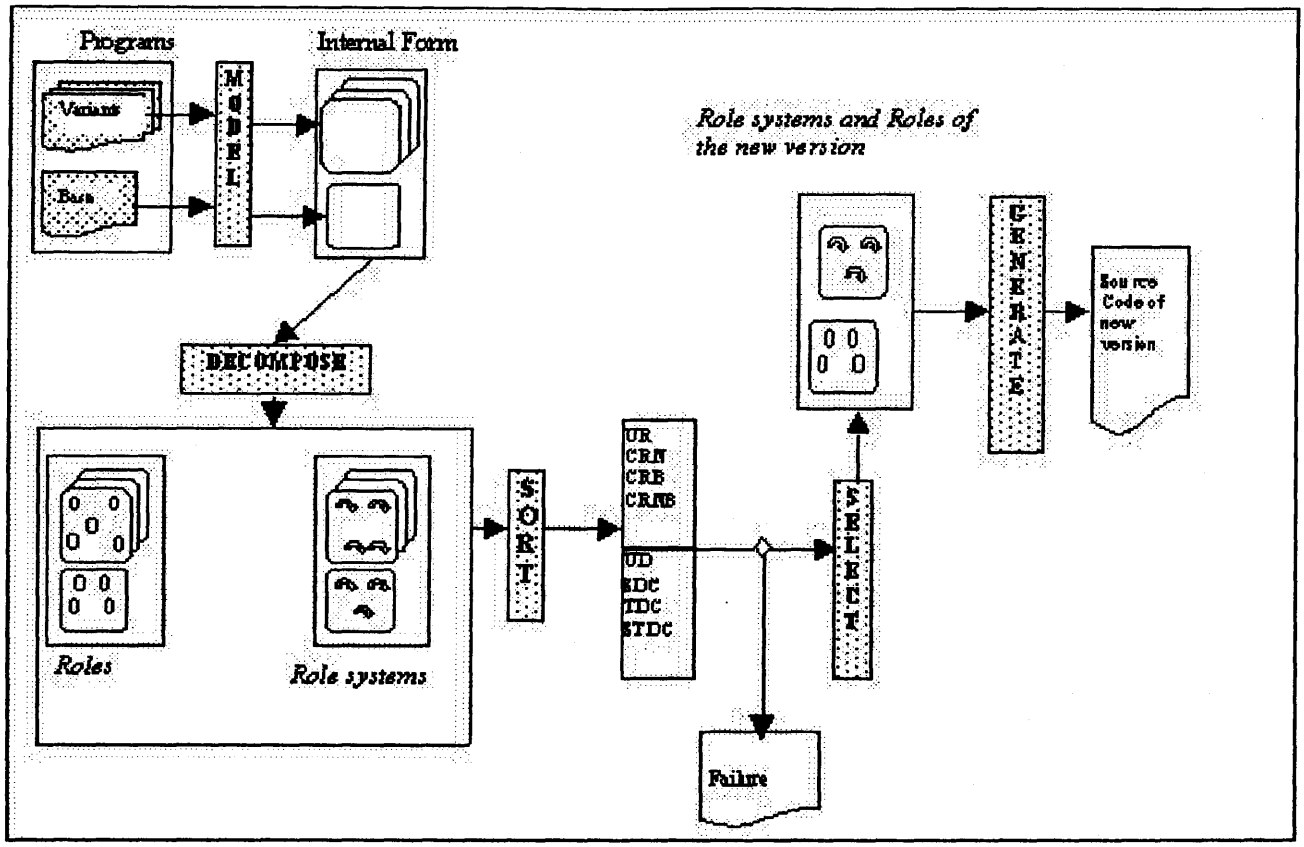


Figure 5: ATLACY architecture

program as many slices as instructions. Their number may be very large. In this article, we have proposed another approach which overcomes these problems. It is based on the concept of a role which models the behaviour with which a variable or a procedure intervenes in a program. This approach has two advantages. The fact that roles do not share instructions permits one to obtain abstraction levels. Also, since there are as many roles as variables and procedures (and not instructions) in a program, the program integration will be made easier.

We plan to complete the model with a help facility to indicate failure in a program integration, or with a recovery on error facility that lists all problem points that must be resolved by human actions.

References

[1] Arnold R.S. Software Reengineering. ed. IEEE Computer Society, 1993, 3-22.

[2] Bendelloul M.S., Bouras Z.E., Ghoul S. and Khammaci T. Assistance to Program Understanding: a model and an algorithm of fragmentation, Gnie Logiciel (France), 45, 13-23, 1997. (in French).

[3] Berzinz V. Software Merging. ed. IEEE Computer Society, 1995.

[4] Binkley D., Horwitz S. and Reps T. Program integration for languages with procedure calls. ACM Trans. on Soft. Eng. and Meth., 4 (1), 310-354, 1995.

[5] Bouras Z.E., Khammaci T. and Ghoul S. Program Understanding: Interprocedural Slicing Extraction. Proc. of Int. Workshop on Process Engineering, MFPE97, Tunis (Tunisia), 79-94, 1997. (in French).

[6] Bouras Z.E., Khammaci T. and Ghoul S. Role System to Software Maintenance, Proc. of 4th African Conf. on Research in Computer Science, CARI'98, Dakar (Senegal), 145-156, 1998. (in French).

[7] Khammaci T., Bouras Z. E. and Bendelloul M. S. Program Understanding Assistance: A Role-based Decomposition. 12th Int. Conf. on Soft. Eng. and Knowledge Eng., SEKE2000, Chicago (USA), July 6-8, 2000 (Accepted).

[8] errante J., Ottenstein K. J. and Warren J.D. The program dependence graph and its use in optimization, ACM Trans. on Prog. Lang. and Sys., 9(3), 319-349, 1987.

[9] Ghoul S., Structures and Methodologies in Software Process Modeling, Ph.D. Dissertation, University of Annaba, Algeria, 1995. (in French).

[10] upta A. Program Understanding Using Slivers An Experience Report. Proc. of Int. Conf. on Software Maintenance, Bari (Italia), 66-71, 1997.

[11] Horwitz S., Prins J. and Reps T. Integrating non-interfering versions of programs. ACM Trans. on Prog. Lang. and Sys, 11(3), 345-387, 1989.

[12] Mayrhauser A. V. and Vans A. M. Hypothesis-Driven understanding Process During Corrective Maintenance of Large Scale Software. Proc. of Int. Conf. on Soft. Maintenance, Bari (Italia), 12-20, 1997.

[13] Reps T. Algebraic properties of program integration. Science of Computer Programming, 17, 139-215, 1991.

[14] Reps T. and Rosay G. Precise Interprocedural Chopping, ACM SIGSOFT Notes, 20(4), 41-52, 1995.

[15] Tip F. A Survey of Program Slicing Techniques. Journal of Prog. Lang., 3(3), 121-189, 1995.

[16] Weiser M. Program Slicing, IEEE Tran. on Soft. Eng. 10(4), 352-357, 1984.

[17] Yang W., Horwitz S. and Reps T. A Program integration algorithm that accommodates semantics-preserving transformations. ACM Trans. on Soft. Eng. and Meth., 1(3), 310-354,1992.

South African Computer Journal

Number 25, August 2000
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 25, Augustus 2000
ISSN 1015-7999

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research notes. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications of Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript
Manuscripts for *review* should be prepared according to the following guidelines:

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
- the title (as brief as possible)
- the author's initials and surname
- the author's affiliation and address
- an abstract of less than 200 words
- an appropriate keyword list
- a list of relevant Computing Review Categories
- Tables and figures should be numbered and titled.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text according to the Harvard. References should also be according to the Harvard method.

Manuscripts accepted for publication should comply with guidelines as set out on the SACJ web page,

`http://www.cs.up.ac.za/sacj`

which gives a number of examples.

SACJ is produced using the LaTeX document preparation system, in particular LaTeX 2_ε. Previous versions were produced using a style file for a much older version of LaTeX, which is no longer supported. Please see the web site for further information on how to produce manuscripts which have been accepted for publication.

Authors of accepted publications will be required to sign a copyright transfer form.

Charges
Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect typesetting, reproduction and other costs. Currently, the minimum rate is R30.00 per final page for contributions which require no further attention. The maximum is R120.00, prices inclusive of VAT.

These charges may be waived upon request of the author and the discretion of the editor.

Proofs
Proofs of accepted papers may be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Letters and Communications
Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words. Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book Reviews
Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement
Placement of advertisements at R1000.00 per full page per issue and R500.00 per half page per issue will be considered. These charges exclude specialised production costs, which will be borne by the advertiser. Enquiries should be directed to the editor.

South African Computer Journal

Number 25, August 2000
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 25, August 2000
ISSN 1015-7999

Contents