VOL. 1, NO. 7, October 2011 ISSN 2222-9833
ARPN Journal of Systems and Software
©2009-2011 AJSS Journal. All rights reserved

http://www.scientific-journals.org

Integrating Non-Interfering Versions of Software Architecture Descriptions

[1]Z. E Bouras, [2]M Maouche
[1]High School of Sciences and Technologies and Sciences, Annaba Algeria, [2]Department of Software Engineering, Faculty of Information Technology, Philadelphia University 19392 Jordan
[1]bourasz@yahoo.com, [2]mmaouch@philadelphia.edu.jo

ABSTRACT

During the last two decades the software evolution community has intensively tackled the program integration issue whose main objective is to merge in a consistent way different versions of source code descriptions corresponding to different and independent variants of the same system. Well established approaches, mainly based on the dependence analysis techniques, have been used to bring suitable solutions. More recently, software evolution researchers focused on the need to develop techniques and tools that support software architecture understanding, testing and maintaining. The objective of this paper is to investigate the software architecture integration, which is a new interesting issue. Its purpose is to merge independent versions of software architecture descriptions instead of source code descriptions. The proposed approach, based on dependence analysis techniques, is illustrated through an appropriate case study.

Keywords: *Software architecture Maintenance, Software Architecture Understanding, Software Architecture Slicing, Software Architecture Integration.*

1. INTRODUCTION

Practical software systems are constantly changing in response to changes in user needs and the operating environment. This arises when new requirements are introduced into an existing system, or specified requirements are not correctly implemented, or the system is to be moved into a new operating environment.

Prior to implementing any change, it is essential to understand the software product as a whole and the programs affected by the change in particular. Software understanding involves the following actions: (1) having a general knowledge of what the software system does and how it relates to its environment, (2) identifying where in the system changes are to be effected; and (3) having an in-depth knowledge of how the parts to be corrected or modified work [1] [2]. The set of changes resulting from these actions must be carefully managed [3].

Software practitioners are used first to manage individually each change in a separate and independent way leading to a new version, then to check that all resulting individual versions do not exhibit incompatible behaviors (non-interference), and finally to merge them into a single version that incorporates all changes (if they do not interfere).

Such techniques, known as program integration techniques, have been widely used at the level of source code using in particularly dependency analysis [4] [5] [6] [7].

Since the emergence of software architecture as a central concept in software engineering, software architecture integration is recognized as an important issue. The development of techniques and tools to support architecture understanding, testing, reengineering, maintaining, and reusing are becoming an important issue [1] [8] [9].

While program integration is appropriate for legacy systems, where software descriptions are often missed, software architecture integration is well suited for architecture centric designed systems.

Only software architecture capturing and understanding issues have been addressed [10] [11] [12] [13]. The objective of this paper is to suggest an approach to deal with the software architecture integration issue. More precisely, we suggest reusing a well known and efficient program integration algorithm [14]. This paper will show the applicability of the approach through an appropriate case study.

The rest of the paper is organized as follows. Section 2 introduces basic concepts software architecture integration. Section 3 presents our software architecture integration process, and finally the section 4 will develop our approach through a complete case study.

2. SOFTWARE ARCHITECTURE INTEGRATION

Practical software systems are constantly changing in response to changes in user needs and the operating environment. This arises when, for instance, new requirements are introduced into an existing software architecture description leading to new software architecture description. Software architecture integration is a process that aims to build a software architecture description that incorporates the new software requirements. It requires a preliminary and primordial step which is known as software architecture understanding [8] [10] [11] [12] [13].

Among the well-known approaches that cope with architecture understanding we can mention the works of Zhao [8], Guo [10], and Rodrigues [11] that are mainly based on

©2009-2011 AJSS Journal. All rights reserved

Dependence Analysis Techniques (DAT). DAT, traditionally used for source code analysis, have been extended to software architectures. We argue, in this paper, that these DAT may be used to cope with the whole software architecture integration process. Before stating our software architecture integration approach, it is useful to introduce some preliminary concepts related to software architecture and their understanding.

2.1. Concepts of Base and Variant software architectures

Nowadays software architecture description is seen as a set of components linked together by means of a set of connectors according to a specific structure (architecture style). Components represent computational entities while connectors represent interaction mechanisms between components [9]. Architecture changes, due to requirement changes, consist of adding/deleting/updating components and connectors. In this paper we distinguish between two kinds of software architecture description: Base software architecture description that represents the original software architecture for which changes are requested and variant software architecture descriptions that represent a family of related and independent versions resulting from changes done on the original software architecture.

2.2. Architecture slicing

Slicing is a technique that allows extracting design elements (components/connectors) of a software architecture that might affect, or be affected by any requirement changes. The primary idea of architecture slicing has been presented in [10] and used in software architecture regression testing [9].

Intuitively, an architecture slice may be viewed as a subset of the behavior of a software architecture description [8, 10]. Its intent is to isolate the behavior of a specified subset of components and/or connectors from the rest of the software architecture description.

Thus, the set of all architecture slices of given software architecture description will cover the whole software architecture behavior. Comparing the behavior of a base software architecture description with the behavior of a variant software architecture description consists of finding architecture slices that are preserved and those that are changed. Changed architecture slices reflect adding/deleting/updating components and connectors.

2.3. Software Architecture Description Graph

The process of architecture slice extraction from the software architecture description is based on the concept of Software Architecture Description Graph (SADG).

SADG consists of representing explicitly, through a graph, dependences between design elements i.e. component-connector, connector-component, and additional dependences. Informally, a SADG is an arc-classified digraph whose (1) vertices represent either the components or connectors in the description, and (2) arcs represent dependencies between architecture design elements. More concretely a slice represents a graph traversal whose origin is a given vertex of interest.

2.4 Software Architecture Interference

[4] [5] [6] [7] identified two kinds of situations that may lead to the existence of incompatibilities between different (source code) variants to be merged. They are referred as “Type I interference” and “Type II interference”. In software architecture description we are only interested by Type I interference. Type I interference occurs when peer slices exhibit different behaviors.

3. SOFTWARE ARCHITECTURE INTEGRATION PROCESS

In this section, we show how to reuse and adapt the Horwitz algorithm, originally devoted to program integration process [14a], in the context of software architecture integration. We argue that this approach solves the issue of architecture integration because both, program and architecture integration, may be brought to a graph theory problem. We limit our work to integration cases where interferences don’t occur.

3.1. Architecture Integration Algorithm

According to Horwitz [14] the integration process achieved, consists of the following steps: (1) start from a **Base** code, (2) build a set of **variants** (resulting from **Base** changes), (3) compare each variant with the base, (4) determine the sets of changed and preserved slices, and (5) combine the variants to form a single integrated new version (if changes don’t interfere). Details of steps (3) and (4) are described below.

Step 3: Compare each variant with the base:

- 3.1. Build the dependency graphs of the software architecture description Base and variants.
- 3.2. Extract, for each dependency graphs, its associated slices.
- 3.3. For each variant
  - 3.3.1. Map each slice of the base software with its variant software peer.
  - 3.3.2. Determine and collect changed and preserved slices.

Step 4: Combine SADGs of Base and variants to form a new SADG.

- 4.1. Check that variants do not interfere
- 4.2. Merge preserved and changed slices.
- 4.3. Derive the resulting dependency graph.
- 4.4. Generate the SADG of the new version of software architecture description from the resulting dependency graph.

Steps 3.1 and 3.2 have already been solved by Zhao [8] in the context of software architecture understanding. Our

©2009-2011 AJSS Journal. All rights reserved

contribution in this paper is to develop the sub-steps 3.3.2/ 4.1/ 4.2/ 4.3 /4.4 in the context of software architecture integration. In the following we will formalize and explicit these last sub-steps.

3.2. Algorithm Formalization

Assuming the following:

$\Delta_{X, Base}$: the set of all changed architecture slices between the variant X and the Base.

$AP_{X,Base}$: the set of vertices whose architecture slices in $SADG_{Base}$ and $SADG_X$ differ.

$PP_{X,Y,Base}$: the set of vertices whose architecture slices in $SADG_{Base}$, $SADG_X$, and $SADG_Y$ are identical.

V ($SADG_x$): set of vertices in SADG of x

$SADG_X/v$: a vertex in the SADG of X from where we want to inspect its impact in the overall SADG.

$b(SADG_X, AP_{X, Base})$: set of peer changed slices when comparing $SADG_{Base}$ and $SADG_X$.

a. Changed slices

Changed architecture slices are computed as the following:

$$AP_{A, Base} = \{v \in V(SADG_A) \mid (SADG_{Base}/v) \neq (SADG_A/v)\}$$
$$AP_{B, Base} = \{v \in V(SADG_B) \mid (SADG_{Base}/v) \neq (SADG_B/v)\}$$
$$\Delta_{A, Base} = b(SADG_A, AP_{A, Base})$$
$$\Delta_{B, Base} = b(SADG_B, AP_{B, Base}).$$

b. Preserved Slices

Preserved architecture slices ($Pre_{A, Base, B}$) are computed as the following:

$$PP_{A, Base, B} = \{v \in V(SADG_{Base}) \mid (SADG_A/v) = (SADG_{Base}/v) = (SADG_B/v)\}.$$
$$Pre_{A, Base, B} = (G_{Base}, PP_{A, Base, B}).$$

c. compatibility between peer slices

Peer slices of $SADG_A$, $SADG_B$ $SADG_M$ if the following hold:

$$b(SADG_M, AP_{A, Base}) \neq b(SADG_A, AP_{A, Base}) \text{ or}$$
$$b(SADG_M, AP_{B, Base}) \neq b(SADG_B, AP_{B, Base}).$$

c. Forming the merged SADG

The merged graph G_M characterizes the SADG of the new version of the software architecture. G_M is computed as the following:

$$G_M = \Delta_{A, Base} \cup \Delta_{B, Base} \cup Pre_{A, Base, B}.$$

4. CASE STUDY

A relevant case study inspired from [13] will illustrate and validate the suggested integration approach. Starting from an initial software architecture description (**Base**) of the suggested case study, we introduce two independent requirement changes that are expected to be compatible (non-interfering). For this purpose two independent copies of **Base** are first created then modified (**Variant A** and **Variant B**). We will proceed as follows:

a. Generate the SADG of Base, Variant A, and Variant B.
b. Extract slices from the SADGs.
c. Determine the set of changed slices and the set of preserved slices.
d. Show that Variant A and Variant B do not interfere.
e. Merge the set of changed slices and the set of preserved slices in order to get the SADG of the new version.

4.1. Case Study Statement

The case study consists of an initial version of a stand alone order management system where: (1) The way of payment (cash, or credit card, etc.) is unspecified, (2) Orders are rejected if some requested items which are not in stock. We assume that our initial software architecture description was designed according to an event based software architecture style. In such style, components intercept and react to a set of relevant events present at their input ports for which they subscribed. As a result, they perform a specific action and produce at their output ports some events to be consumed by other components or the external environment. Components are connected by means of connectors that link specific output ports with specific input ports.

The system involves five components interconnected as in Figure 1:

(1)- Order_Req_Handler: plays the role of interface with the order clerk.
(2)- Order_Entry: extracts the requested items from orders for furthering processing.
(3)- Inventory: checks the availability of requested items in the stock. If some items are not found, orders will be rejected.
(4)- Shipping: takes care of gathering the ordered items.
(5)- Accounting: accumulates the total amount of money for the order.

Notice that the unspecified way payment is depicted by the external connection of **Accounting** component (**Accounting_res_out** to **Accounting_res_in**). For sake of simplicity, details related to components ports and events are omitted. More details can be found in [13]. Figure 1 depicts the Base software architecture description of the order management system.

©2009-2011 AJSS Journal. All rights reserved

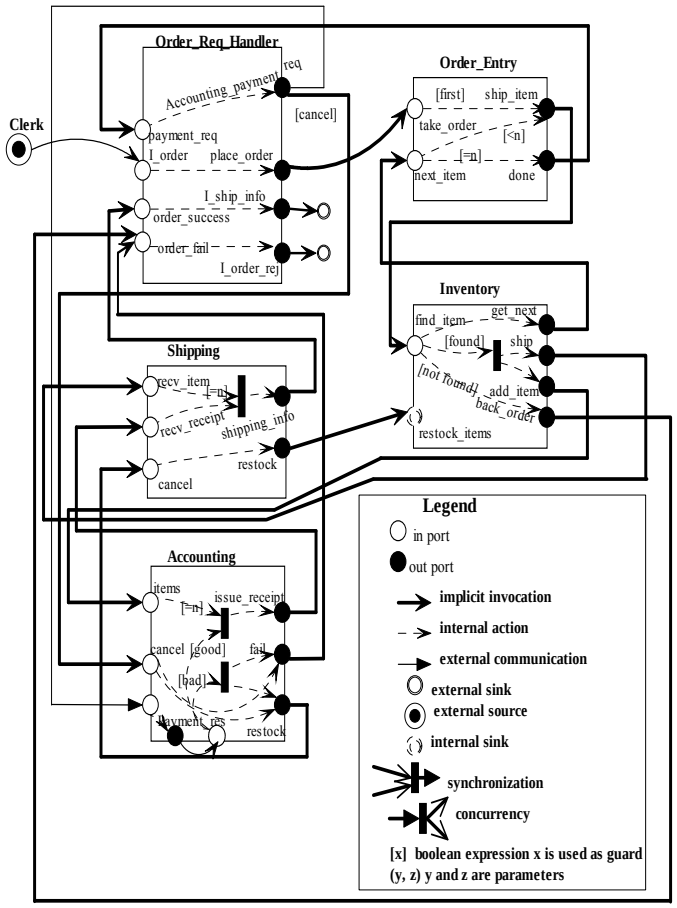

Figure 1. Base software architecture description of the order management system.

4.2. Building the SADG of variants

Two non-interfering variants are considered. Variant A: a credit card payment option is added. Variant B: in case stocks are empty at the order time, the request is handled through a back order mechanism.

4.2.1. SADG of Variant A

In Variant A, a credit card payment option is requested. Software Architect **A** inserts a new component that will take in charge this functionality. This leads to the following changes in the architecture description of the software: (1) adding a new component (**Credit_Checker**), (2) creating new connectors (from **Order_Req_Handler** to **Credit_Checker**, from **Credit_Checker** to **Accounting**), and (3) updating existing connections: removing external connection (from **Credit_res_out** to **Credit_res_in**). Figure 2 illustrates the SADG of this variant.

4.2.2. SADG of Variant B

In Variant B, a back order mechanism is requested. Software Architect **B** inserts a new component that will take in charge this functionality. This leads to the following changes in the architecture description of the software: (1) adding a new component (**Back_order**), (2) creating new connectors (from **Inventory** to **Back_order**, from **Back_order** to **Accounting**, from **Back_order** to **Shipping**). Figure 3 represents the SADG of variant B.

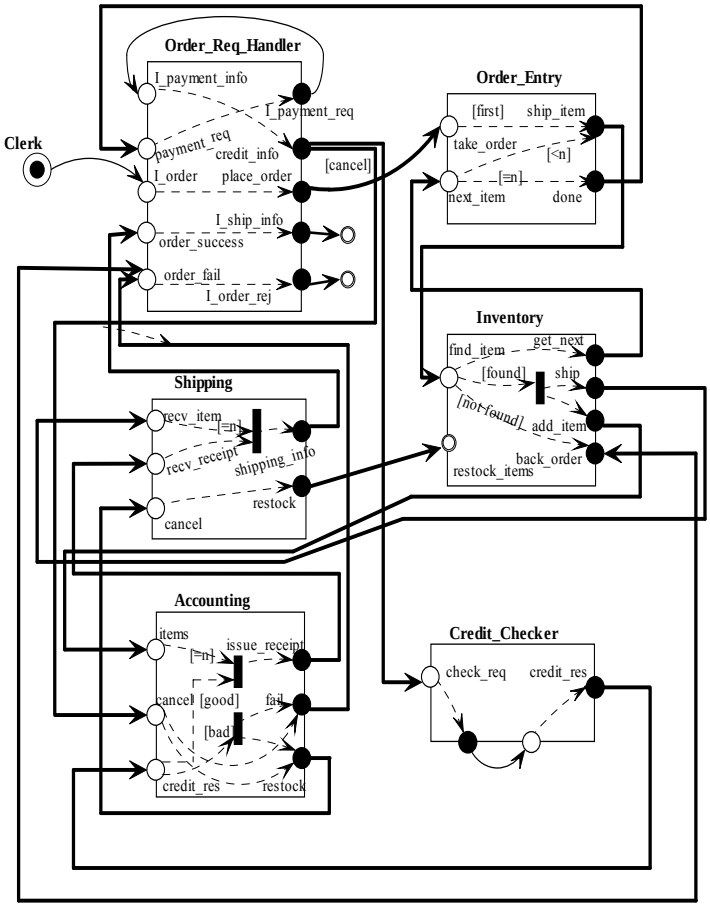

Figure 2. SADG of variant A.

©2009-2011 AJSS Journal. All rights reserved

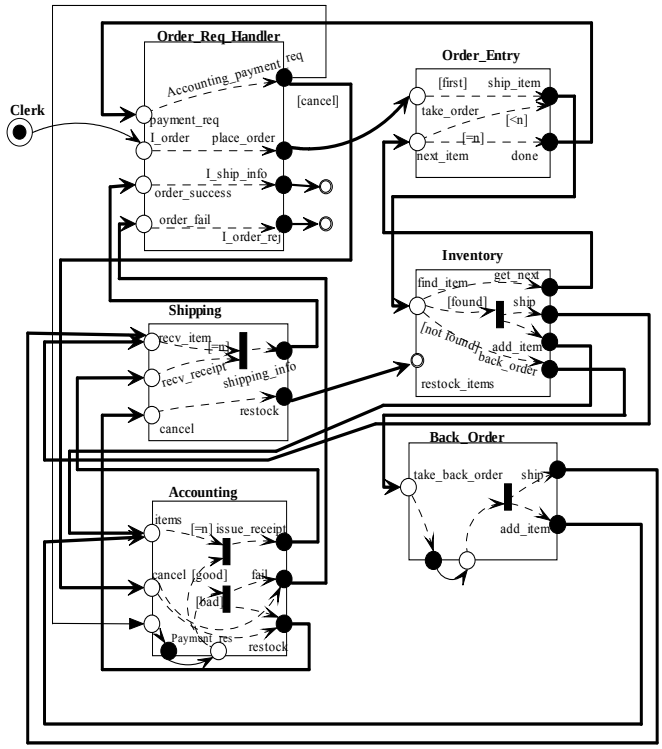

Figure 3. SADG of variant B.

4.3. Slices Extraction

The extraction process outcomes more than fifteen slices per SADG. For lack of space, only a pertinent sample of computed slices is presented in this paper. The rest of slices are given in an internal report.

The selected sample involves one example of *changed slices* case (Base/Variant A) and one example of *preserved slices* case (base, Variant A, and Variant B). These examples focus on the following two behaviors of interest: slice traversals that leads to the canceling of orders (payment unspecified and credit payment), slice traversal that leads to a successful ordering. Table 1 gives more details on the selected slices.

Table 1: Sample of slices

Slice Nb	Source	Behavior of interest	Slice (traversal)
1	Base	canceling of orders (payment unspecified)	Order_Req_Handler:I_order, Order_Req_Handler:place_order, Order_Entry:take_order, Order_Entry:ship_item, Inventory:find_item, Inventory:get_next, Order_Entry:next_item, Order_Entry:done, Order_Req_Handler:payment_req, Order_Req_Handler:I_Payment_req, Order_Req_Handler:I_payment_info, Order_Req_Handler:credit_info, Accounting:credit_res, Accounting:credit_res_out, Accounting:credit_res_in Accounting:fail, Order_Req_Handler:order_fail, Order_Req_Handler:I_order_rej.
2	Variant A	canceling of orders (credit payment)	Order_Req_Handler:I_order, Order_Req_Handler:place_order, Order_Entry:take_order, Order_Entry:ship_item, Inventory:find_item, Inventory:get_next, Order_Entry:next_item, Order_Entry:done, Order_Req_Handler:payment_req, Order_Req_Handler:I_Payment_req, Order_Req_Handler:I_payment_info, Order_Req_Handler:credit_info, **Credit_Checker:check_req, Credit_Checker:check_req_out, Credit_Checker:check_req_in, Credit_Checker:credit_res,** Accounting:credit_res, Accounting:credit_res_out, Accounting:credit_res_in Accounting:fail, Order_Req_Handler:order_fail, Order_Req_Handler:I_order_rej.
3	Base, Variant A, Variant B	successful ordering	Order_Req_Handler:I_order, Order_Req_Handler:place_order, Order_Entry:take_order, Order_Entry:ship_item, Inventory:find_item, Inventory:get_next, Order_Entry:next_item, Order_Entry:next_item, Order_Entry:ship_item, Inventory:find_item, Inventory:ship, Inventory: add_item, Accounting:items, Accounting:issue_receipt, Shipping:recv_item, Shipping:shipping_info, Order_Req_Handler:order_success, Order_Req_Handler:I_ship_info.

Comments:

- Slice 1 and Slice 2, that are peer, display different graph traversal (depicted in bold). Thus they will be classified in the category of changed slices.
- Slice 3 reflects the same behavior in the three SADG. Thus they will be classified in the category of preserved slices.
- Intersection of changed slices between Base and Variant A and changed slices between Base and Variant B gives an empty set that is there is no interference between changes.

4.4. Forming the merged SADG

This step involves forming a new SADG by using the result of steps 4.2 and 4.3. This consists of merging all changed architecture slices between SADG of Base and variants, and thus preserved in the three SADGs. The union of changed and preserved slices forms the SADG of the new version of software architecture description. Figure 4 depicts the SADG of the new version of software architecture description.

©2009-2011 AJSS Journal. All rights reserved

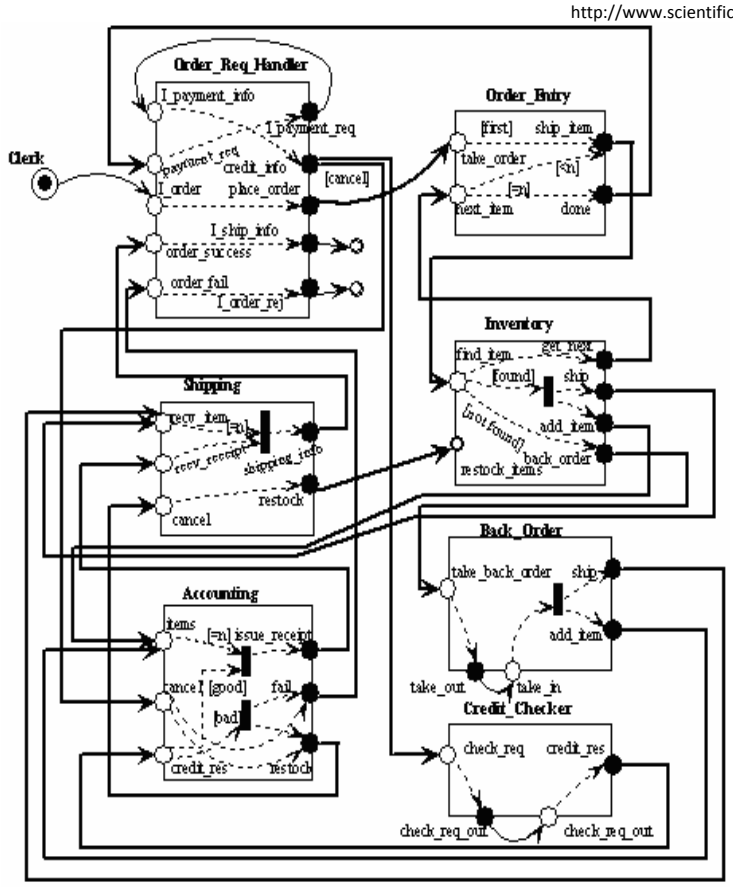

Figure 4. SADG of the new version of software architecture description.

CONCLUSION

In this paper we have shown that the concept of Software Architecture Slicing, initially defined to cope with Software Architecture Understanding, may be also used to bring the Software Architecture Integration issue. A concrete case study illustrated the suggested approach.

We are planning to consolidate this approach with others realistic case studies involving Interfering and Non-Versions of Software Architecture Descriptions.

Another promising investigation consists of tackling the Software Architecture Integration where software architecture are described by well-known Architecture Description Languages (ADLs)

REFERENCES

1. Penny G, Takang A**.** *Software Maintenance: Concepts and Practice*. World Scientific Publishing Co**,** 2003.

2. Barais O. *Software Architecture Evolution, Software Evolution*, Springer-Verlag: Berlin Heidelberg 2008; 233-263.

3. Westhuizen C, Van der Hoek A. *Understanding and Propagating Architectural Changes.* Proceedings of the Working IFIP Conference on Software Architecture; 2002.

4. Horwitz S, Prins J, Reps T. *Integrating non-interfering versions of programs.* ACM Transactions on Programming Languages and Systems 1989**;** 11(3): 345-387.

5. Binkley D, Horwitz S, Reps T. *Program Integration for Languages with Procedure Calls*. ACM Transactions on Software Engineering. and Methodology 1995, 4, (1): 3-35.

6. Bouras Z, Ghoul S., Khammaci T. *A new approach for program Integration.* The South African Computer Journal 2000, 25: 3-11.

7. Khammaci T, Bouras Z. *Versions of Program Integration*. Handbook of Software Engineering and Knowledge Engineering, **2**: World Scientific Publishing: Singapore 2002.

8. Zhao J., Xiang L. *Architectural Slicing to Support System Evolution*, Idea Group Publishing, 2005, ISBN 1-59140-368-5, 197-210.

9. Jaiprakash T, Lalchandani R. *Regression testing based-on slicing of component-based software architectures*. Proceedings of the 1st conference on India software engineering conference, Hyderabad, India, 2008; 67-76.

10. Guo J, Liao Y, Pamula R. *Static Analysis Based Software Architecture Recovery.* Computational Science and Its Applications - ICCSA 2006, Lecture Notes in Computer Science, 2006, 3982/2006, 974-983.

11. Rodrigues F, Barbosa S. *Component Identification Through Program. Slicing.* Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005), Electronic Notes in Theoretical Computer Science, 160, 2006, 291-304.

12. Jaiprakash T. Lallchandani, Mall R. *Static Slicing of UML Architectural Models.* Journal of Object Technology, 2009, 8 (1), 159-188.

13. Kim T, Song Y, Chung L. *Software architecture analysis: a dynamic slicing approach Source.* ACIS International Journal of Computer & Information Science 2000, 1(2).

14. Horwitz S, Reps T. *The use of dependence graph in software engineering*. Proceedings of the 14th International on software engineering, Melbourne, Australia 1992.