Software Architectures Evolution Based Merging

Zine-Eddine Bouras
LISCO Laboratory, Department of Mathematics and Computer Sciences
P.O. Box 218, EPST Annaba Algeria
E-mail: z.bouras@epst-annaba.dz

Mourad Maouche
Department of Software Engineering, Faculty of Information Technology
P.O. Box 1 Philadelphia University 19392, Jordan
E-mail: mmaouch@philadelphia.edu.jo

Keywords: software architecture, software architecture merging, sependency analysis, slicing

Received: November 12, 2015

During the last two decades the software evolution community has intensively tackled the software merging issue. The main objective is to compare and merge, in a consistent way, different versions of software in order to obtain a new version. Well established approaches, mainly based on the dependence analysis techniques on the source code, have been used to bring suitable solutions. However the fact that we compare and merge a lot of lines of code is very expensive. In this paper we overcome this problem by operating at a high level of abstraction. The objective is to investigate the software merging at the level of software architecture, which is less expensive than merging source code. The purpose is to compare and merge software architectures instead of source code. The proposed approach, based on dependence analysis techniques, is illustrated through an appropriate case study.

Povzetek: Prispevek se ukvarja z ustvarjanjem nove verzije programskega sistema iz prejšnjih na nivoju abstraktne arhitekture.

1 Introduction

Software evolution is the response to software systems that are constantly changing in response to changes in user needs and the operating environment. This arises, often, when new requirements are introduced into an existing system, specified requirements are not correctly implemented, or the system is to be moved into a new operating environment [1]. One way to cope with evolution is to carry out the software from the scratch, but this solution is very expensive. Another way, that is less expensive, is to proceed by merging changes.

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) [2].

Such techniques, known as program merging, have been widely used at the level of source code [2-4]. However comparing and merging a huge number of lines of codes is very expensive. Our main motivation is to overcome this problem by going up at the level of software architecture where the number of comparison and merge is smaller than in the source code.

In this way, we must address some problems like (1) understanding what an existing architecture does and how it works (dependency analysis), (2) how to capture the differences between several versions of a given architecture, and (3) how to create new architecture. The first problem was resolved by Kim et al. [5].

The objective of this paper is to suggest an approach to deal with the rest of problems, namely finding an approach to compare and merge software architectures. More precisely, we suggest reusing the well-known and efficient program merging algorithm due to Horwitz [6]. This paper will show the applicability of this algorithm through an appropriate example.

The rest of the paper is organized as follows. Section 2 is dedicated to related works. Section 3 presents the notion of software architecture description and a running example to be used throughout this paper. Section 4 introduces software merging in general and software architecture merging in particular. Section 5 is dedicated to the needed concepts in our approach. In section 6 we present, detail, and illustrate our approach of software architecture description merging.

2 Related Works

Besides differencing programs done by Horwitz [6], there are other works that investigate differencing hierarchical information for a large code such that Apiwattanapong et al. in [7] and Raghavan et al. in [8].

In the context of design differencing Xing and Stroulia in [9] use the assumption that the entities they are differencing are uniquely named and many nodes match exactly. A basic change due to designers is to rename entities in order to become more expressive. In this way the proposed approach fails.

Abi-Antoun et al. in [10] propose an algorithm based on empirical evaluation to cope with architectural merging issue. Empirical Evaluation losses information in some cases, merging architecture needs the study of dependencies, formally, between components.

Finally there is an approach that copes with software architectures evolution based merging. Bouras and Maouche [11] use an internal form to represent software architecture and proceed by a syntactic differentiation. They, also, detect some type of conflicts that can fail the process.

Our approach is more formal and precise in term of dependency analysis. It uses the technique of slicing that is a formal filter. Slicing permits dependency analysis of software architecture by allowing us to find matching's and differences between elements of Software Architecture Descriptions (SAD) during merging process, and then merge components (if they are compatibles) to obtain a new version of SAD.

3 Software architecture description

Understanding all aspects of complex system is very hard. It therefore makes sense to be able to look at only those aspects of a system that are of interest at a given time. The concept of architecture views exists for this purpose. According to IEEE 2007, a view is a representation of a whole system from the perspective of a set of concerns. Each view addresses a set of system concerns, following the conventions of its viewpoint, where a viewpoint is a specification that describes the notations and modeling techniques to be used in a view to express the architecture in question from a given perspective [12]. Examples of viewpoints include: Functional viewpoint, Logical viewpoint, Component-and-connector viewpoint, etc. This paper is based on Component-and-connector viewpoint which specify structural properties of component and connector models in an expressive and intuitive way. They provide means to abstract away direct hierarchy, direct connectivity, port names and types, and thus can crosscut the traditional boundaries of the implementation-oriented hierarchical decomposition of systems and sub-systems [13, 14].

3.1 The example: Electronic Commerce

We introduce the running example, inspired from [5], to be used throughout this paper.

An order entry form is entered, electronically by a clerk. This form is taken by the Electronic Order Processing System (EOPS) and transformed on several actions through its five components: Ordering, Order_Entry, Inventory, Shipping, and Accounting. Components are distributed over different platforms, have a number of connectors between them, are independent processes, and communicate with each other through parameterized events. EOPS is depicted in figure 1.

EOPS stores the order information through CGI, and triggers Ordering which is the front-end of the whole system. This triggering is done through an I_order event. I_order generates a place_order event (internal action depicted by a dotted arrow) at the place_order port. The payment results from a payment_req event of Ordering which takes place when Ordering gets notified from the Order_Entry (implicit invocation depicted by a bold arrow).

When the payment gets approval, Ordering gets an order_success event and generates I_ship_info event to notify the customer of a successful order (internal action). Otherwise Ordering gets an order_fail event and notifies customer of unsuccessful order through I_order_rej event (internal action).

Order_Entry gets a take_order event from Ordering whenever customer places an order (external communication depicted by an arrow). An order is broken down into several items and each information of them is sent to Inventory through a ship_item event along with the customer information. ship_item events are generated whenever each ordered item is processed by Inventory to pass next item information until all the items for an order are processed. The done event results from a next_item event when there is no more items to be processed and this event triggers the payment information request payment_req of Ordering. Inventory generates a get_next event whenever it gets a find_item event to get the other item information for the order. Inventory generates two events, a ship event to Shipping and add_item to Accounting if an item is in the inventory it generates a back_order event to Inventory in order to get and ship the out-of-stock item, otherwise (concurrency). A restock_items event occurs when a customer cancels an order, this event does not cause any further event generation and is represented by special symbol called internal sink.

Shipping takes care of gathering the items of an order through recv_item events from Inventory. When it gets a shipping approval through a recv_receipt event from Accounting, it generates a shipping_info event to Ordering and it ships ordered items (synchronization). When it receives a cancel event (due to canceled order), it generates a restock event along with the item information it received. Accounting accumulates the total amount for an order whenever it receives an items event and verifies resources by communicating with outside components when it receives a checking request and sends the result (e.g., good/bad). Upon receiving payment_res (i.e., good or bad), it issues either an issue_receipt event as an approval for shipping when successful or fail and restock events to inform the failure of the order process to the customer.

4 Software architecture merging

Merging approaches take a form when concurrent modifications of the same system are done by several developers. They are able to merge changes in order to obtain ultimately one consolidated version of a system

again. However, they are faced to two challenges: the representation and how to find out differentiation.

The first one concerns the representation of software artifact on which the merge approach operates. It may either be text-based or graph-based. The second challenge concerns how differences are identified, represented, and merged [14, 15].

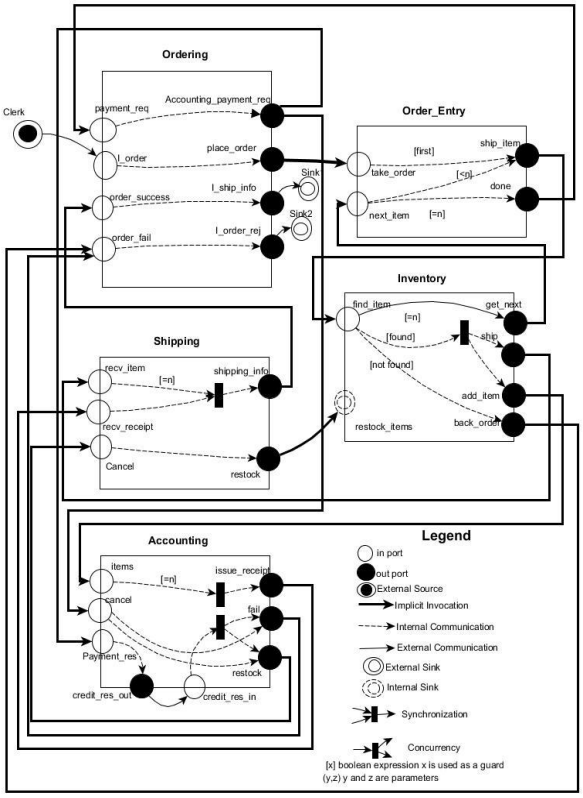

Figure 1: Software architecture of electronic order processing system.

Text-based merge approaches operate solely on the textual representation of a software artifact in terms of text files. The unit element of the text file may either be a paragraph, a line, a word, or even an arbitrary set of characters. Unit element of given version is compared to the original unit element in order to create the new one. The major advantage of such approaches is their independence of the programming languages used in the versioned artifacts. However, the major problem when merging flat files, syntax and semantics of a programming language are losses [14, 15].

Graph-based approaches overcome these problems; they operate on a graph-based representation of a software artifact for achieving more precise merging. Such approaches translate the versioned software artifact into a specific structure (graph) before merging. The unit elements (e.g. components) are represented by nodes and their relationships (e.g. connectors) by arcs. Changes consist of adding/deleting/updating unit elements [16]. However it requires a preliminary and primordial step which is known as software architecture understanding [16, 17]. It is very important to understand component's context and its running environment in order to efficiently manage all kinds of dependencies. In general, as soon as a new component is installed, removed, or updated in a given software architecture, it has an impact on a part of the system. The new component may refer to certain components, and also be used by other components [16-20].

5 Software architecture merging concepts

Before starting our software architecture merging approach, it is useful to introduce some preliminary concepts related to software architectures and their understanding. These concepts concern how to represent software architecture as a graph (Software Architectural Description Graph**)** and how to find matching's and differences between components (slicing), and finally merging them.

5.1 Software Architectural Description Graph

Understanding a software addresses some problems like what it does and how it works. This is due to the implicit relationships between lines of codes. An explicit representation is needed.

Kim et al. in [5] propose a suitable dependence graph to support SAD named Software Architectural Description Graph (SADG). It consists of representing explicitly, dependencies between architecture elements i.e. component-connector, connector-component, and additional dependences. Informally, SADG is an arc-classified digraph whose vertices represent either the components or connectors in the description, and arcs represent dependencies between architectural design elements. Formal definitions and illustrations of SADG are detailed in [5].

In this paper we distinguish between two kinds of software architectural description: Base and variants. Base represents the original software architecture for which changes are requested. Variants represent a family of related and independent versions resulting from changes done on Base by independent developers. Also we point out that merge conflicts may occur. They take place if one change invalidates another change, or if two changes do not commute. Then, it is not decidable where to integrate changes [3]. For example if software architect of Variant A decides to update boolean expression [=n] to [n=10] in component Ordering_Entry (between in port next_item and out port done), while software architect of Variant B states that the same n will be n= 20, we are in the front of a conflict between architects, and merging process fails. In this case conflict is resolved manually. In this paper, we consider merging architectures without conflicts.

5.2 Architectural slicing

When a maintenance programmer wants to modify a component in order to satisfy new requirements, the programmer must first investigate which components

will affect the modified component and which components will be affected by the modified component. By using a slicing method, the programmer can extract the parts of a software architecture containing those components that might affect, or be affected by, the modified component. This can assist the programmer greatly by providing such change impact information.

Using architectural slicing to support change impact analysis of software architectures promises benefits for architectural evolution. Slicing is a particular application of dependence graphs. Together they have come to be widely recognized as a centrally important technology in software engineering. This due to the fact they operate on the deep rather than surface structures, they enable much more sophisticated and useful analysis capabilities than conventional tools [6].

Traditional slicing techniques cannot be directly used to slice software architectures. Therefore, to perform slicing at the architecture level, appropriate slicing notions for software architectures must be defined with new types of dependence relationships using components and connectors. Some works have investigated the issue of adapting the definition PDG to the level of software architecture. Between them, we can cite works of Rodrigues and Barbosa in [17] which propose the use of software *slicing* techniques to support a component's identification process via a specific dependence graph structure, the FDG (Functional Dependency Graph).

Zhao's technique, in [21] is based on analyzing the architecture of a software system given in Acme ADL. He captures various types of dependencies that exist in an architectural description. The considered dependencies arise as a result of dependence relationships existing among ports and/or roles of components and/or connectors. Architecture slicing technique operates by removing unrelated components and connectors, and ensures that the behavior of a sliced system remains unaltered.

Kim introduced an architectural slicing technique called dynamic software architecture slicing (DSAS) in [5]. A dynamic software architecture slice represents the run-time behavior of those parts of the software architecture that are selected according to the particular slicing criterion of interest to the software architect such as a set of resources and events.

An important distinction between a *static* and a *dynamic* slice is that static slices is computed without making assumptions regarding inputs, whereas the computation of dynamic slice relies on a specific test case. In other words, the difference between static and dynamic slicing is that dynamic slicing assumes *fixed* input, whereas static slicing does not make assumptions regarding the input, hence smaller in size than its static counterpart.

In order to illustrate the concept of dynamic slicing consider the fact that, we are interested by the run-time behavior of those parts of the software architecture of EOPS that are selected according to the particular slicing criterion when a customer wants to sell only one item that is in the inventory. The dynamic slicing concept is dedicated to find all implied parts of EOPS.

This triggering is done through an I_order event. I_order generates a place_order event at the place_order port. The payment results from a payment_req event of Ordering which takes place when Ordering gets notified from the Order_Entry.

Order_Entry gets a take_order event from Ordering whenever customer places the order (wiyh n=1). The item is sent to Inventory through a ship_item event along with the customer information. ship_item event is generated whenever the ordered item is processed by Inventory. Inventory generates a ship event to Shipping.

Shipping takes care of gathering the items of an order through recv_item events from Inventory. It generates a shipping_info event to Ordering and it ships ordered items. Finally Ordering gets an order_success event and generates CGI_ship_info event to the CGI program to notify the customer of a *successful* order. The run-time behavior is depicted in Figure 2.

Our graph representation is inspired from Kim's researches [5] where the process of architectural slice extraction from the software architectural description is based on the concept of Software Architectural Description Graph (SADG) and is a graph traversal.

Finally, comparing the behavior of Base with the behavior of a given variant consists of comparing static architectural slices of Base with static architectural slices of the given variant.

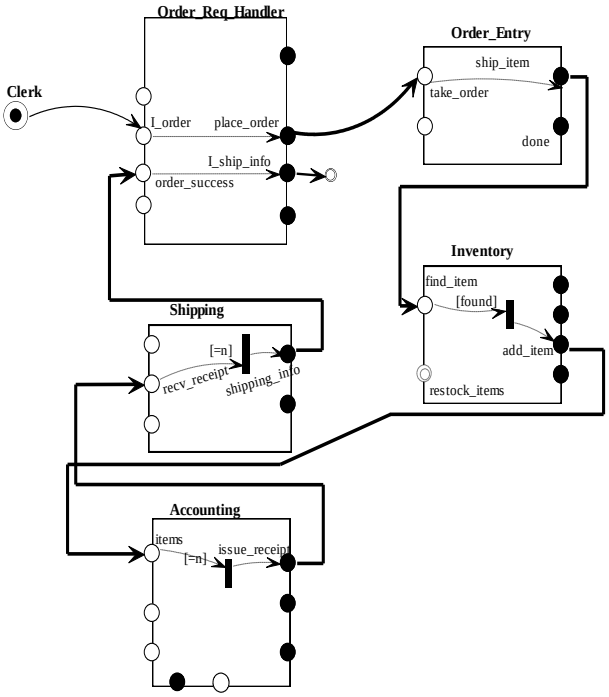

Figure 2: Example of Architecture Dynamic Slice.

5.3 Graph similarities

Comparing two graphs needs at first to find, for a given node (or edge) in a given graph, its corresponding node (or edge) in the other. An efficient way to find out similarity is the use of signature and structural matching [16].

A signature is defined as a pair of corresponding elements needs to share a set of properties such as type

information, which can be a subset of their syntactical information. Type information can be used to select the elements of the same type from the candidates to be matched because only elements with the same type need to be compared. Signature is used as the first criterion to match elements as proposed by [16]. If there is more than one candidate that has been found, the signature cannot identify a node uniquely. It is, therefore, to do further analysis by structural matching.

Structural matching is based on calculation of Graph Similarity using Maximum Common Edge Subgraphs [16]. The first algorithm to find the candidate node with maximal edge similarity for a given host node takes the host node and a set of candidate nodes of graph 2 as input, computes the edge similarity of every candidate node and returns a candidate with maximal edge similarity. The second algorithm for computing edge similarity between a candidate node and a host node takes two maps as, input, stores all the incoming and outgoing edges of the host and candidate nodes indexed by their edge signature. By examining the mapped edge pairs between these two maps, the algorithm computes the edge similarity as output. Graph similarities algorithm can be summarized as the following:

Let Base and a Variant SADG's

1. For each variant node

    1.1 Use signature matching to find candidate node
        If there is more than one candidate use structural matching
        Compare each node and its associated edges of Base with its variant peer (similar).

    1.2 Determine and collect sets of changed elements
        If no candidate, host node belongs to Delete set
            // exists in Base and not in Variant
        Remaining nodes in variant belongs to New set
            // exists in Variant and not in Base
        Compare names of each pair of nodes mapping
        If values are different, name belongs to Update set
            // all node mapping and differences are found

2. Edges connecting to delete nodes are Delete edges
    Edges connecting to new nodes are New edges
    Apply signature matching to find out the edge mapping
    Remaining edges in Base belongs to Delete edges set
    Remaining edges in variant belongs to New edges set

All nodes in N1 have been examined by signature and structural matching; all possible node mappings between N1 and N2 are found.

6 Software architecture merging process

In this section we show how to reuse and adapt the Horwitz algorithm [6] to the context of software architecture merging. We show that this approach solves the issue of architecture merging because both, program and architecture merging may be brought to a graph theory problem.

6.1 Software merging algorithm

Figure 3 resumes merging process. It starts from (1) a Base Software Architecture Description, (2) build a set of variants (resulting from Base changes), (3) build Software Architectural Description Graph for each SADG, (4) compare each variant with Base to determine sets of changed and preserved elements, and (5) combine these sets to form a single integrated new version (if changes don't interfere). Steps (1) and (2) are done concurrently by developers, in step (3) we construct SADG's according to Kim's approach.

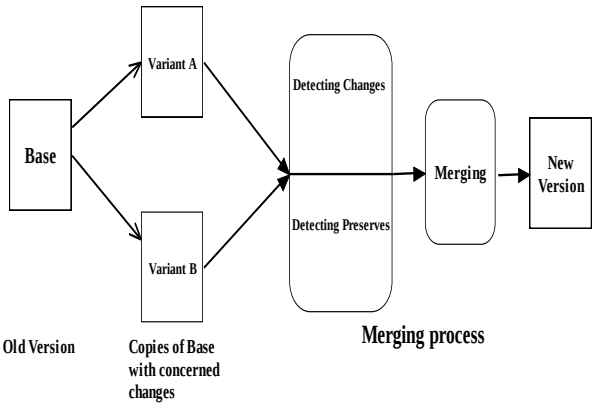

Figure 3: Merging Process.

Step 4: Compare each variant with the base to determine sets of changed and preserved elements

For each variant

4.1. Determine peer nodes and edges with Base by signature and structural matching.

4.2. Extract from each SADG the associated slices.

4.3. Determine sets of changed and preserved elements

4.3.1. Map and compare each slice of the base software with its peer in variant.

4.3.2. Determine and collect changed and preserved slices.

Step 5: Combine changed and preserved slices to form a new SADG.

5.1. Merge preserved of Base and changed slices of variants.

5.2. Check that variants do not interfere

5.3. Derive the resulting dependency graph.

5.4. Generate the SADG of the new version of software architecture description from the resulting SADG.

Our contribution in this paper is to develop steps (4) and (5) in order to merge software architecture. In the following we formalize these sub-steps.

6.2 Formalization

Given SADGs $SADG_{Base}$, $SADG_A$, and $SADG_B$, of Base, and variants A and B respectively. The algorithm performs three steps.

The first step identifies three subgraphs that represent the changed behavior of A with respect to Base(ΔA, Base), the changed behavior of B with respect to Base (ΔB, Base) and the preserved behavior that is the same in all architectures (PreA,B,Base) by using the set of vertices whose slices in SADGBase, SADGA, and SADGB are identical (i.e. . PPA,B,Base).

The second step unifies these subgraphs to form a merged dependence graph SADGM.

In the third step, a merged architecture GM is generated from graph SADGM.

6.2.1 Construction of a slice

First, we show how to compute an architecture slice. In this section we use the notation Component_name: inport/outport_name in order to represent components and connectors in an internal form. For example, Order_Req_Handler:I_order is the input port I_order of component Order_Req_Handler.

Each SADG is transformed in an internal form. The internal form is a set of triplets (a, b, c) which reflects the fact that there is an edge of type c from a to b. c can be an implicit invocation (ii), an internal action (ia) or an external communication (ec) while a and b are components and connectors using the previous notation.

For example (Ordering:I_order,Ordering:place_order,ia) means that: there is an internal action (ia) between in port I_order of component Ordering (Ordering:I_order) and out port place_order of Ordering (Ordering:place_order)

Table 1 represents a sample of internal form of Base SADG.

Architecture	Internal Form
Base	((External_Source_Clerk, Ordering:I_order,ec), (Ordering:I_order,Ordering:place_order,ia), (Ordering:payment_req, Ordering:I_payment_req, ia), (Ordering:order_success, Ordering:I_ship_info,ia), (Ordering:order_fail, Ordering:Iorder_rej,ia), (Ordering: I_payment_req, Ordering:payment_req,ec), (Ordering: I_ship_info, sink1, ec), (Ordering:I_payment_req,Accounting:items,ii), (Ordering:place_order,Order_Entry:take_order, ii), ……)

Table 1: A sample of internal form of Base SADG.

Because of we are interested by static dependency analysis of SADG, we extract all slices starting from external source entry (e.g. clerck) to component that is in the front-end of the whole system until the end of the process (e.g. external sink). A static slice is a graph traversal by transitive closure from external source node to a final node from where we cannot continue the traversal (e.g. external sink).

In our example of Figure 1 there are more than fifteen slices that represent the complete behavior of EOPS. Table 2 represents one of them.

Slice	Internal form
	((External_Source_Clerk, Ordering:I_order), (Ordering:I_order,Ordering:place_order,ia), (Ordering:place_order,Order_Entry:take_order,ii), (Order_Entry:take_order, Order_Entry:ship_item,ia), (Order_Entry:ship_item, Inventory:find_item,ii), (Inventory:find_item, Inventory:get_next,ia), (Inventory:get_next, Order_Entry:next_item,ii), (Order_Entry:next_item, Order_Entry:done,ia), (Order_Entry:done, Ordering:payment_req,ii), (Ordering:payment_req, Ordering:Accounting_payment_req,ia), (Ordering, Accounting _payment_req, Accounting:cancel,ii), (Accounting:cancel, Accounting:fail,ia), (Accounting:fail, Ordering:order_fail,ii), (Ordering:order_fail, Ordering:I_order_rej, ia), (Ordering:I_order_rej, Sink2,ec))

Table 2: Internal form of a static slice.

Note that this slice reflects the behavior of canceling an order.

At the end of this step, each one (Base and variants) SADG's is transformed into a set of slices and the process of comparison can starts.

6.2.2 Changed slices

Let $\Delta_{X,\ Base}$ the set of changed slices between variant X and Base. Changed slices are computed as the following:

$$AP_{A,\ Base} = \{v \in V(SADG_A) \mid (SADG_{Base}/v) \neq (SADG_A/v)\}$$

$$AP_{B,\ Base} = \{v \in V(SADG_B) \mid (SADG_{Base}/v) \neq (SADG_B/v)\}$$

$$\Delta_{A,\ Base} = b(SADG_A, AP_{A,\ Base})$$

$$\Delta_{B,\ Base} = b(SADG_B, AP_{B,\ Base}).$$

Where

$V(SADG_x)$ denotes the set of vertices in SADG of variant X.

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

$b(SADG_X, AP_{X,\ Base})$ is the set of peer changed slices in $SADG_{Base}$ and $SADG_X$.

In other words, internal forms of peer slices are compared. As a result they haven't the same graph traversal (different internal forms).

An example of changed slices is introduced in the section dedicated to application (6.3).

6.2.3 Preserved slices

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

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

$$Pre_{A,\ Base,\ B} = (SADG_{Base}, PP_{A,\ Base,\ B}).$$

The same graph traversal exists in both Base and variants.

We find an example of preserved slices in the section of application.

6.2.4 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:

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

Informally, GM is composed of slices that are changed in SADG's of variants A and B with respect to Base and those that are unchanged.

6.3 Application

In this section we illustrate and validate the suggested merging approach through the running example of figure 1.

Starting from an initial software architecture description (Base) we introduce two independent requirement changes that are expected to be compatible. For this purpose two independent copies of Base are first created and modified concurrently (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 these SADG's.
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.

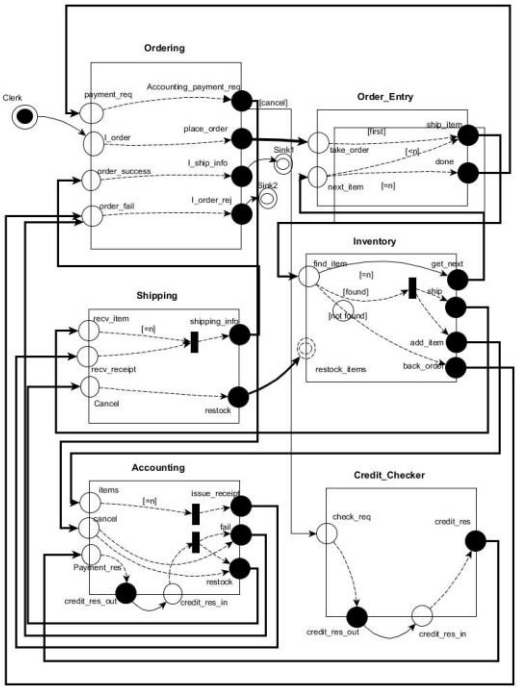

Figure 4: SADG of variant A.

6.4 Building SADG's of variants A and B

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

6.4.1 Variant A SADG

In Variant A, Software Architect A inserts a new component that will take in charge the credit card payment option. This leads to the following changes in the architectural description of the software: (1) adding a new component (Credit_Checker), (2) creation of new connectors (from Ordering to Credit_Checker, and from Credit_Checker to Accounting), and (3) removing external connection (from Credit_res_out to Credit_res_in). Figure 4 represents the SADG of variant A.

6.4.2 Variant B SADG

In Variant B, Software Architect B inserts a new component that will take in charge the back order mechanism. This leads to the following changes in the architectural description of the software: (1) adding a new component (Back_order), (2) creation of new connectors (from Inventory to Back_order, from Back_order to Accounting, from Back_order to Shipping). Figure 5 depicts the SADG of variant B.

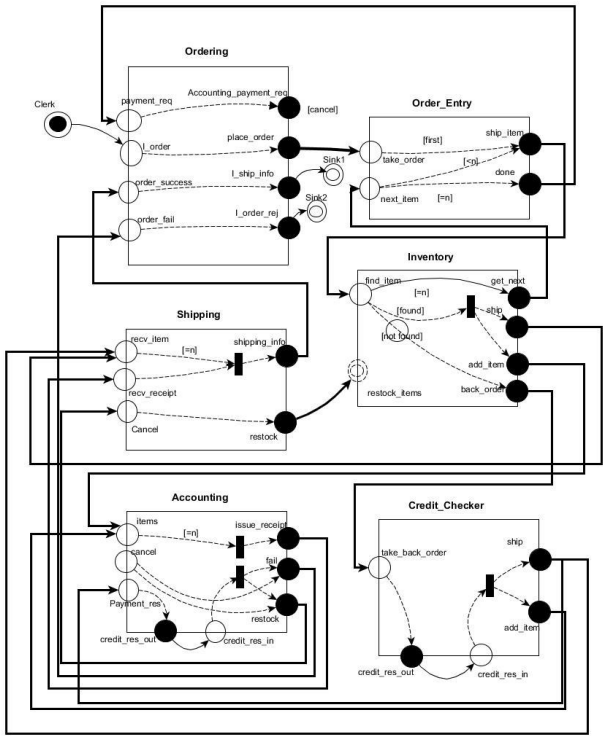

Figure 5: SADG of variant B.

6.5 Slice Extractions

Slice 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. Selected sample involves an example of *changed slices*

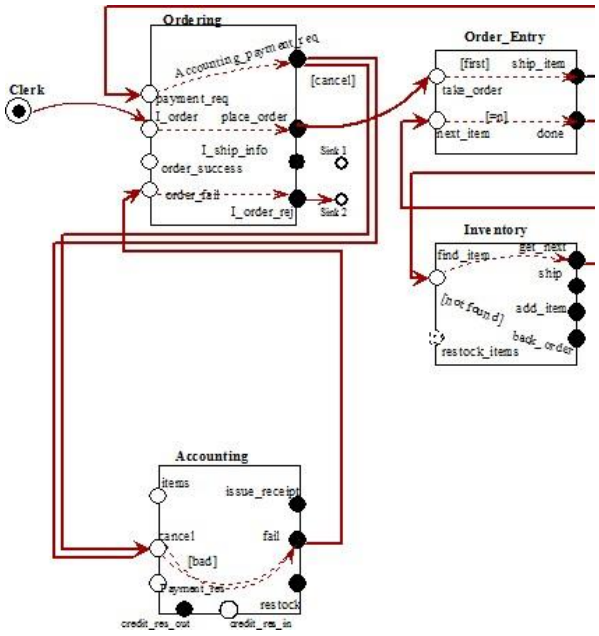

Figure 6: Slice of canceling order of Base.

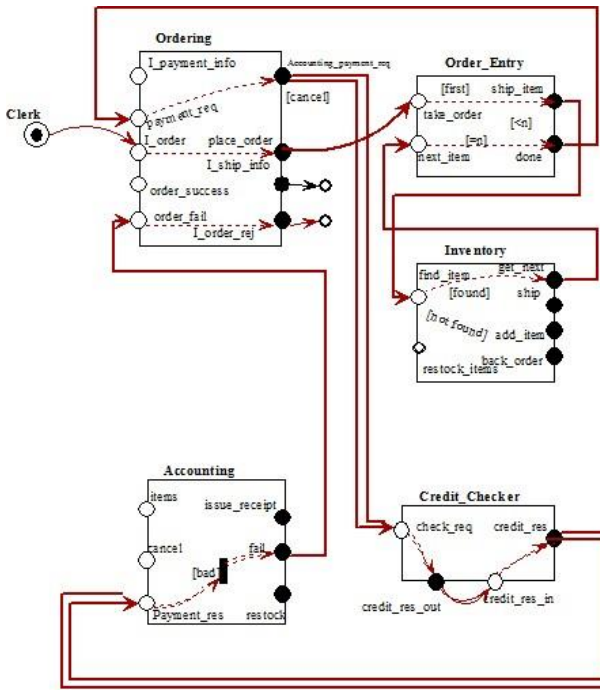

Figure 7: Slice of canceling order of Variant A.

.case ($\Delta_{A,\ Base}$ = $b(SADG_A, AP_{A,\ Base})$ and an example of *preserved slices* case ($Pre_{A,\ Base,\ B}$ = $(G_{Base}, PP_{A,\ Base,\ B})$). These examples focus on the following two behaviors of interest: (1) slice traversals that leads to the canceling orders (payment unspecified and credit payment), and (2) slice traversal that leads to a successful ordering.

Figures 6 and 7 illustrate changed slice of canceling order in Base and Variant A respectively. Differences between these peer slices are depicted with double arrows in figures 6 and 7. In this case the slice of Variant A will belong to ΔA, Base set and is one of slices forming the SADG of the new version of Software architecture.

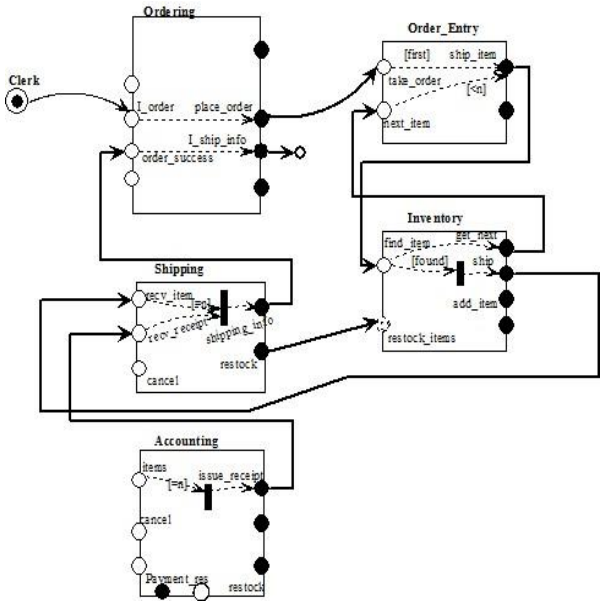

Figure 8: Slice of successful ordering in Base, Variants A and B.

Figure 8 reflects the same behavior in the three SADG. The graph traversal of successful ordering slice is the same in Base and variants.

Thus they will be classified in the category of preserved slices. They belong to:

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

Intersection of changed slices between Base and Variant A and changed slices between Base and Variant B gives an empty set, consequently there is no interference between changes. We can continue the process.

6.6 Forming the merged SADG

This step involves forming a new SADG by using the result of previous steps. It consists of merging all changed architectural slices between SADG of Base and variants, and thus preserved in these SADGs. The union of changed and preserved slices forms the SADG of the new version of software architectural description.

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

Figure 9 depicts the SADG of the new version of software architectural description.

7 Conclusion

First, it is important to situate our works according to Horwitz's works.

The main contribution of Horwitz's work was to propose a new process of software evolution. This process was formalized and implemented at the program level. So Horwitz opened a new way for future research in evolution through the life cycle of software. This way involves mainly the facts (1) make explicit the

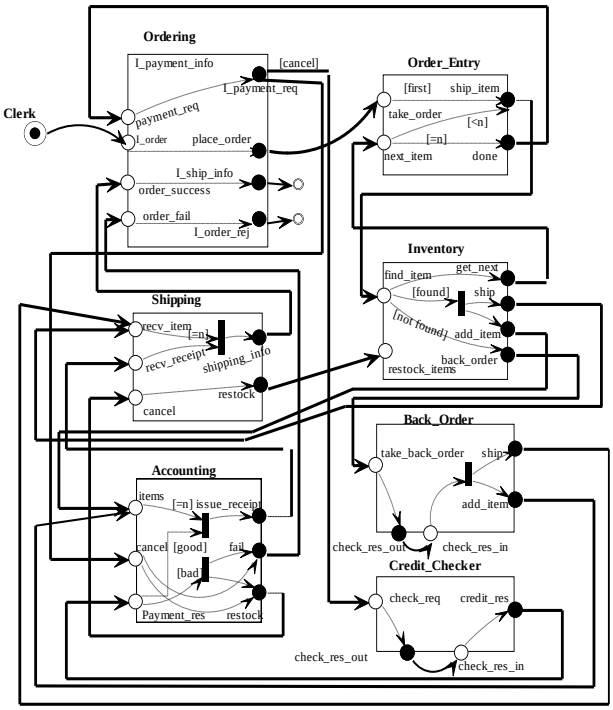

Figure 9: SADG of the new version.

dependencies between elements (e.g. data and control) which are usually implicit, (2) extract all behaviors (influence of one element over the other) for each version (Variants and Base), and (3) compare the behavior of each variant according to the Base program and finally form the new version that consists of the elements that remained preserved in all versions and those that have created differences in the variants.

Since, several studies have been made by exploiting this process. We also followed this process, but at the level of software architectures. We solved the problem of the similarity of graphs, ignored by Horwitz. We investigate and found the best way to represent the dependencies between elements of architectures that are different than those of programs. From there we followed the process. So, we showed that software evolution based merging at the level of software architecture is possible. Consequently this will lessen the cost of evolution.

Nowadays we continue in the theoretical aspects of this approach. Particularly, we are planning to investigate, consolidate and implementing this approach by involving conflicts. Another promising investigation consists of tackling the Software Architecture Merging where software architectures are described by well-known Architecture Description Languages (ADLs). Indeed in some cases architectures are provided in terms of ADLs. The question is "is it possible to merge architectures from ADLS or passing, first, by the graph transformation?"

References

[1] Tom Mens (2008). "Introduction and Roadmap: History and Challenges of Software Evolution". Eds Tom Mens · Serge Demeyer, Springer-Verlag Berlin Heidelberg, pp. 1-14.

[2] T. Mens (2002), "A State-of-the-Art Survey on Software Merging", IEEE Transactions on Software Engineering, vol 28, no 5, pp. 449–462.

[3] D. Binkley, S. Horwitz, and T. Reps (1995). "Program Integration for Languages with Procedure Calls", ACM Transactions on Software Engineering and Methodology, vol 4, no 1, pp. 3-35.

[4] T. Khammaci, and Z. Bouras (2002). "Versions of Program Integration", Handbook of Software Engineering and Knowledge Engineering, vol 2, World Scientific Publishing: Singapore, pp. 465-486.

[5] T. Kim, Y. Song, and L. Chung (2000). "Software architecture analysis: a dynamic slicing approach", International Journal of Computer & Information Science, vol 1, no 2, pp. 91-103, 2000.

[6] S. Horwitz, and T. Reps, "The use of dependence graph in software engineering", Proceedings of the 14th International on software engineering, Melbourne, Australia, 1992, pp. 392-411.

[7] T. Apiwattanapong, A. Orso, and M. Harrold (2004), "A Differencing Algorithm for Object-oriented Programs", Automated Software Engineering, vol. 14, no 1, pp. 3-36.

[8] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine (2004). "A semantic-graph differencing tool for studying changes in large code bases" Proceedings of 20th IEEE International Conference on Software Maintenance, pp. 188-197.

[9] Z. Xing, and E. Stroulia (2005), "UMLDiff: An Algorithm for Object–Oriented Design Differencing", Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), pp. 54-65.

[10] M. Abi-Antoun, J. Aldrich, N. Nahas, B. Schmerl and D. Garlan (2006). "Differencing and Merging of Architectural Views", proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06), pp. 47-58.

[11] Z. Bouras, M. Maouche (2015). "Merging software architectures with conflicts detections", International Journal of Information Systems and Change Management, Eds. Inderscience Publisher, Vol 7, No 3, pp. 242-260.

[12] K. Kobayashi, M. Kamimura; K. Yano; and K. Kato (2013). "SArF map: Visualizing software architecture from feature and layer viewpoints", in Proceedings of International Conference on Program Comprehension (ICPC'2013), San Fransisco USA, 2013, pp. 43 – 52.

[13] S. Maoz, J. Ringert, and B. Rumpe (2013). "Synthesis of Component and Connector Models from Crosscutting Structural Views", Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13), Eds. B. Meyer, pp. 444-454.

[14] B. Westfechtel (2010). "A Formal Approach to Three-Way Merging of EMF Models", Proceedings of the 1st International Workshop on Model Comparison in Practice, IWMCP '10, Malaga, Spain, pp. 31-41.

[15] P. Brosch, M. Seidl, M. Wimmer and G. Kappel (2012). "Conflict Visualization for Evolving UML Models", Journal of Object Technology, vol 11, no 3, pp. 1–30.

[16] P. Langer, K. Wieland, M. Wimmer, and J. Cabot (2011), "From UML Profiles to EMF Profiles and Beyond", eds. TOOLS. LNCS, vol. 6705, Springer, Heidelberg, pp. 52–67.

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

[18] J. Guo, Y. Liao, and R. Pamula (2006). "Static Analysis Based Software Architecture Recovery", Computational Science and Its Applications Lecture Notes in Computer Science, 3982, pp. 974-983.

[19] B. Li, Y. Zhou, Y. Wang, and J. Mo (2005). "Matrix-based component dependence representation and its applications in software quality assurance" SIGPLAN Notices, vol 40, no 11, pp. 29–36.

[20] J. Lalchandani (2009). "Static Slicing of UML Architectural Models", Journal of Object Technology, vol 8, no 1, pp. 159-188.

[21] J. Zhao (2000). "A Slicing-Based Approach to Extracting Reusable Software Architectures," Proceedings of the. 4th European Conference on Software Maintenance and Reengineering, IEEE Computer Society Press, Zurich, Switzerland . pp.215-223.