

Software Architecture Analysis: A Dynamic Slicing Approach

Taeho Kim, Yeong-Tae Song, Lawrence Chung and Dung T. Huynh
Dept. of Computer Science
The University of Texas at Dallas
email: {tkim, ysong, chung, huynh}@utdallas.edu

Abstract

As the complexity of software systems increases, so does the need for a good mechanism of abstraction. Software architecture design is an abstraction, hiding an immense amount of details about the data structures, algorithms, idiosyncrasies of programming language constructs, etc. that may be used in implementing the system-to-be. Fundamental as it may be to the modeling of the system, the very nature of this high level abstraction can also pose difficulties with the understanding and analysis of the behavior of the system-to-be. This paper introduces the notion of *dynamic software architecture slicing (DSAS)* in order to alleviate such difficulties. A dynamic software architecture slice represents the run-time behavior of those parts of the software architecture that are selected according to a particular slicing criterion such as a set of resources and events. This paper also describes a methodology for using the notion, and an algorithm to generate dynamic software architecture slices. The feasibility and the expected benefits of the approach is demonstrated through a study of part of an electronic commerce system and a run-time execution of its architecture using a tool.

Keywords: Software Architecture, Dynamic Slicing, E-Commerce, Event-driven

1 Introduction

As the complexity of software systems increases, so does the need for a good mechanism of abstraction. This is especially true in the presence of the various emerging paradigms, such as the component-based software engineering paradigm in which systems are built out of existing systems or their parts. Software architecture design is an abstraction which defines a software system mostly in terms of components which carry out computations, control and data storage, of connectors which are used by the components to interact with each other, and of constraints that are imposed on the behavior of the components and connectors [Shaw96]. Currently standing as the highest level of solution during software development, an architecture hides an immense amount of

details about the data structures, algorithms, idiosyncrasies of programming language constructs, etc. that may be used in implementing the system-to-be.

Fundamental as it may be to the modeling of the system, the very nature of this high level abstraction can also pose difficulties with the understanding and analysis of the behavior of the system-to-be. One reason is that an architecture is a generic description which entails potentially an infinite number of different system behaviors. Another reason perhaps is due to the description of the system behavior at the architecture level, which is often non-trivial to precisely specify and to easily understand.

This paper introduces the notion of *dynamic software architecture slicing (DSAS)* in order to alleviate such difficulties. 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. This paper also describes a methodology for using the notion, and a forward dynamic software architecture slicing algorithm to generate dynamic software architecture slices. The feasibility and the expected benefits of the approach is demonstrated through a study of part of an electronic commerce system and a run-time execution of its architecture using a tool.

The notion of DSAS draws on earlier work on dynamic program slicing techniques [17, 1, 10, 9] and static software architecture slicing techniques [20, 16]. While a static slice is determined independently of the input at compile time, a dynamic slice is determined according to a particular input at run time, hence smaller in size than its static counterpart. In contrast to a program slice which represents the set of program statements that are relevant to the particular variables of interest at some point during the program execution, a software architecture slice is a set of (parts of) architecture components and connectors that are relevant to the particular variables and events of interest at some point during the architecture level execution.

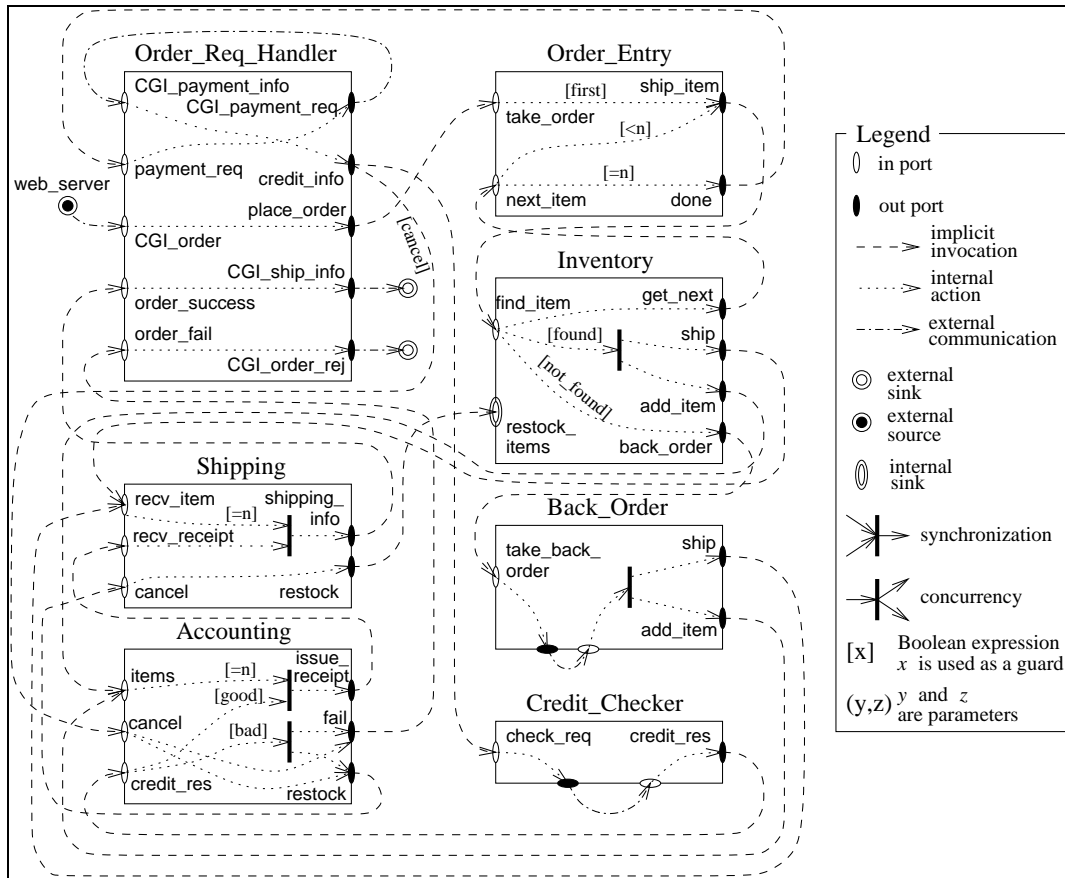


Figure 1 Electronic-Commerce architecture diagram

The role of DSAS is to isolate a particular execution path at the architecture level. In order to demonstrate the feasibility and the expected benefits of the approach, this paper shows as an example a study of a small portion of an Electronic Commerce System, in particular the use of a tool in generating traces of events pertaining to specific variable-value assignments and the events in the slicing criterion.

Section 2 presents a portion of an Electronic Commerce System along with a set of basic definitions of DSAS. Section 3 describes the DSAS methodology and algorithm. Section 4 shows the use of the DSAS methodology in the study of the Electronic Commerce System and how to generate DSAS. Section 5 discusses the DSAS approach in relation to other works. A summary of contributions and future work are given at the end.

2 Software Architecture Dynamic Slicing

In this section, we introduce a running example to be used throughout this paper and some basic definitions of software architecture and forward dynamic software architecture slice.

2.1 The Example: Electronic Commerce

A key component of any *e-commerce* system is electronic order processing system [18]. In a typical electronic order processing system (hereafter EOPS), a customer places an order (e.g., software, books or movie-on-demand) electronically by filling in an order entry form. The form is taken by EOPS and results in the generation of a concrete order form, which is to be used for the purpose of accounting, ordering goods or services (from other agents), and shipping.

The software architecture of EOPS, depicted in Figure 1¹, consists of seven components, which are distributed over different platforms, and a number of connectors

¹ We consider interactions between the software architecture and the external (invisible) components. The *inport*, which does not cause any event generation, is called an *internal event sink* and is denoted as a “double oval”. The *external event source* generates events from outside of the architecture and *external event sink* receives events from the architecture. These are depicted as “filled double circle” and “double circle”, respectively.

between the components. The components of EOPS are independent processes that communicate with each other through parameterized events.

The components of EOPS are: order request handler (Order_Req_Handler), order entry (Order_Entry), inventory (Inventory), back order (Back_Order), credit checker (Credit_Checker), shipping (Shipping) and accounting (Accounting).

We assume that web_server has accepted the order information from the customer, stored it through CGI, and triggers Order_Req_Handler which is the front-end of the whole system. This triggering is done through an CGI_order event. CGI_order has customer ID, ordered items and number of each items as parameters and generates a place_order event at the place_order port².

For example, when Order_Req_Handler receives a CGI_order event along with the customer ID and ordered item information as parameters (e.g., customer ID is John, 2 china sets, 1 kitchen table cover and 3 silverware sets, etc.), it generates a place_order event along with the same parameters as in CGI_order to fulfill the customer's order. The CGI_payment_req event results from a payment_req event (to get the credit card information) of Order_Req_Handler which takes place when Order_Req_Handler gets notified from the Order_Entry. This causes the CGI program to get the credit card information from the customer and the information is fed through a CGI_payment_info event. The credit card information provided by the customer is sent either to the Credit_Checker accompanied by the credit_info event or, if the customer decides to cancel the order, sent to Accounting with "cancel" instead. Note that the relationship between a generation of a CGI_payment_req event and reception of a CGI_payment_info event is not internal to the architecture. That is, there is an external component to the architecture (e.g., CGI program) that is not a component of the software architecture itself and is not visible. When the credit check gets approval, Order_Req_Handler gets an order_success event and generates CGI_ship_info event to the CGI program to notify the customer of a *successful* order. Otherwise Order_Req_Handler gets an order_fail event and notifies customer of *unsuccessful* order through a CGI_order_rej event.

Order_Entry gets a take_order event from Order_Req_Handler whenever customer places an order. An order is broken down into several items and

each item information is sent to Inventory through a ship_item event along with the customer information. A 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 item to be processed and this event triggers the payment information request, payment_req of Order_Req_Handler. 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 Back_Order in order to get and ship the out-of-stock item, otherwise. A restock_items event occurs when a customer cancels an order or the credit of the customer is not approved by Credit_Checker. Note that this event does not cause any further event generation and we use special symbol called *internal sink* for that.

Upon receiving a take_back_order event, Back_Order communicates with outside component (e.g., manufacturer or supplier) and sends ship and add_item events to Shipping and Accounting respectively. Shipping takes care of gathering the items of an order through recv_item events from Inventory or from Back_Order. When it gets a shipping approval through a recv_receipt event from Accounting, it generates a shipping_info event to Order_Req_Handler and it ships ordered items. When it receives a cancel event (due to bad credit or 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. Upon receiving credit_res (i.e., *good* or *bad* and *approved* or *rejected*), it issues either an issue_receipt event as an approval for shipping when credit check is successful or fail and restock events to inform the failure of the order process to the customer and prevent shipping by Shipping.

Credit_Checker verifies credit by communicating with outside components when it receives a check_req and sends the result (e.g., *good/bad* or *approved/rejected*) through a credit_res event to Accounting.

2.2 Basic Definitions

We now define the basic concepts of *software architecture* and *architecture slice*.

Definition 1: A software architecture can be defined as

$$A = (C, P, \Delta, \Gamma, \Psi, \Pi, \Omega)$$

² Our convention is that the name of an event is identical with that of the port where the event is received or transmitted.

where

- C is a finite set of *components*.
- $P = P_i \cup P_o$ is a finite set of *ports* where P_i and P_o are finite sets of *imports* and *exports*, respectively. Each port is denoted by a unique port name followed by a set of parameters $\varphi \in \Psi$. The set of *imports* of a component C is denoted by P_i^C and the set of *exports* of a C is denoted by P_o^C . An *event* $e \in E$ refers to the invocation of a *Port*. Unless there is confusion, however, we use the terms *event* and *port* interchangeably.
- $\Delta \subseteq P_o \times P_i$ is a finite set of *connectors*, where each connector $\delta \in \Delta$ can be associated with a *guard* $\gamma \in \Gamma$ and a set of operations $\omega \in \Omega$, i.e., $\Delta = \{ \langle P_o, P_i \rangle^{\omega[\gamma]} \mid p_o \in P_o, p_i \in P_i, \gamma \in \Gamma \text{ and } \omega \in \Omega \}$.
- Γ is a finite set of *guards*. Each guard, $\gamma \in \Gamma$, is a Boolean expression and is enclosed by a pair of angular brackets ($[$ and $]$).
- Ψ is a finite set of *parameters* in the form of $x:T$, where x is a variable of type T .
- $\Pi \subseteq P_i \times P_o$ is a finite set of *internal paths*, where each $\pi \in \Pi$ can be associated with a *guard* $\gamma \in \Gamma$, i.e., $\Pi \subseteq \{ \langle P_i, P_o \rangle^{[\gamma]} \mid p_i \in P_i, p_o \in P_o, \text{ and } \gamma \in \Gamma \}$.
- $\Omega = \left\{ \begin{array}{c} \leftarrow \rightarrow \\ \cup \\ \rightarrow \leftarrow \end{array} \right\}$

where each symbol denotes *concurrency* or *fork* and *synchronization* or *join*, respectively.

For example, the software architecture depicted in Figure 1 has:

$C = \{ \text{Order_Req_Handler, Order_Entry, Inventory, Shipping, ...} \}$
 $P = \{ \text{CGI_Order, credit_info, take_order, ship_item, ship, fail, ...} \}$
 $\Delta = \{ \langle \text{ship_item, find_item} \rangle^{(c,i,n)}, \langle \text{credit_info, cancel} \rangle^{(c)[\text{cancel}], ...} \}$
 $\Gamma = \{ [\text{not_found}], [\text{good}], [\text{bad}], ... \}$
 $\Psi = \{ c, i, n, amt, ... \}$
 $\Pi = \{ \langle \text{payment_req, CGI_payment_req} \rangle, \langle \text{find_item, back_order} \rangle^{[\text{not_found}], ...} \}$

Δ and Γ are represented by dashed and dotted arrows, respectively in the architecture diagram.

Definition 2: Software architecture slice can be defined as $S_A = (C', P', \Delta', \Gamma', \Psi', \Pi', \Omega')$ where C' is a subset of C where the set of ports of a component in C' is a subset of $P_i^C \cup P_o^C$. P' is a subset of *ports*, $P' \subseteq P$. Δ' is a subset of *connectors*, $\Delta' \subseteq \Delta$. Γ' is a subset of *guards*, $\Gamma' \subseteq \Gamma$. Ψ' is a subset of *parameters*, $\Psi' \subseteq \Psi$. Π' is a subset of *internal paths*, $\Pi' \subseteq \Pi$. Ω' is a subset of *concurrencies* and *synchronizations*, $\Omega' \subseteq \Omega$.

2.3 Software Architecture Slicing Criterion

In software architecture slicing, we are interested primarily in *components* and *connectors* instead of program statements. In modeling software architecture, occurrences of events are often the main concern especially when the behavioral characteristic of the architecture is determined by the causality of events.

Let E be a finite set of events generated by the software architecture. We define dynamic *software architecture slicing criterion* as follows.

Definition 3: A dynamic software architecture slicing criterion can be defined as

$$C_d^A = (I_{sc}, e_{sc}, n_{sc})$$

where

$$I_{sc} = \{ \langle v, c \rangle \mid v : T \in \Psi \}$$

c is a value of type T which is bound to v , is a set of input values --- e.g., initial value of a variable (parameter) v_k is c_k ($v_k^0 = c_k$), $e_{sc} \in E = P$ is an event to be observed, and $n_{sc} \in \mathbf{N}$ is an event counter, where \mathbf{N} is *natural numbers*. *Event counter* is usually set to one but when loop or cycle is involved with the occurrence of events, it plays a significant role.

Informally, *software architecture slicing criterion* can be described as: I_{sc} provides input values to the ADL executable. DSAS algorithm computes the slice of the given event e_{sc} until the event occurs the same number of times as n_{sc} in the slicing criterion and the slicing criterion is said to be “satisfied” at this point. The value of n_{sc} can be generalized to “*” in case all the relevant paths of the architecture are of concern.

3 The DSAS Methodology and Algorithm

Dynamic software architecture slicing (DSAS) is a technique to decompose software architecture with respect to the given *slicing criterion*. We describe the methodology and algorithm in this section.

3.1 The DSAS Methodology

The computation of *forward dynamic software architecture slice* using the DSAS method is depicted in Figure 2.

The *forward dynamic* method implies that the architecture slice is computed during run-time. The software architecture slicer needs a run-time environment which takes as input an executable architecture and a set of initial conditions for execution, etc. More specifically, the constituents of DSAS method are as follows:

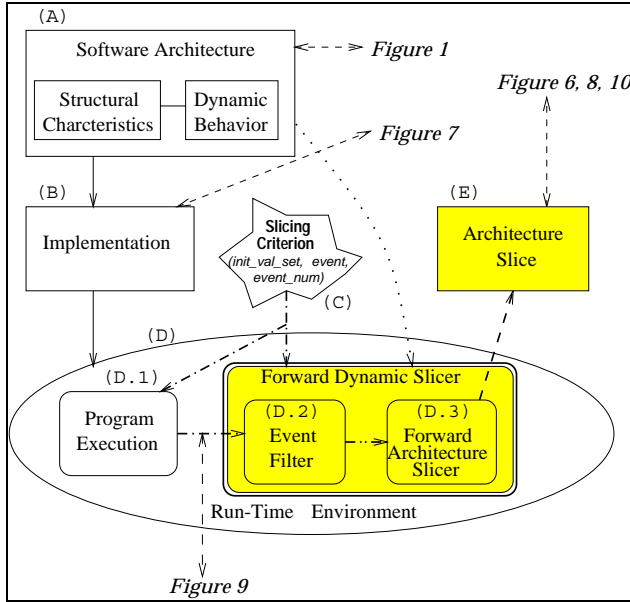


Figure 2 Dynamic software architecture slicing methodology

(A) Software Architecture: A software architecture is designed by software architects using an ADL of choice such as ACME [6], RAPIDE [11], Aesop [5], UniCon [15], and Wright [2]. An architecture consists of *structural* and *behavioral* parts and can be represented by an architecture diagram (e.g., as shown in Figure 1).

(B) Implementation of Software Architecture: The software architect implements the design by mapping the behavioral part of the design into program statements, while retaining the structural properties.

(C) Slicing Criterion: A *slicing criterion* provides the basic information such as the initial values and conditions for the ADL executable, an event to be observed, and occurrence counter of the event. More specifically, a *slicing criterion* consists of three elements: (i) i_{sc} , input values for the ADL executable program, (ii) e_{sc} , an event name and (iii) n_{sc} , a number of occurrences of the event.

(D) Run-time Environment: During run-time, *Forward Dynamic Slicer* gets a *slicing criterion* as input, and reads the ADL source code of the architecture to identify component and connector information along with the event names used in the ADL and parameter names associated with those events.

(D.1) Program Execution: Once an initialization finishes, *Forward Dynamic Slicer* executes the ADL executable, and the ADL executable begins to generate a set of partially ordered events (or *poset* [11, 14]) along with the component and connector information.

(D.2) Event Filter: When *Forward Dynamic Slicer* receives events from the ADL executable, *Event Filter* filters out the events that are not relevant to the software architecture and passes only those events to *Forward Architecture Slicer* that are relevant to the *slicing criterion*.

(D.3) Forward Architecture Slicer: *Forward Architecture Slicer* computes an architecture slice dynamically by examining the components and ports, according to the given slicing criterion, that are visited by those filtered events and the conditions that trigger the events.

(E) Resulting Software Architecture Slice: When the slicing criterion is satisfied, the slice computed up to the event of interest, i.e., e_{sc} , is the resulting *forward dynamic software architecture slice*. Architecture slice is a subset of the architecture that consists only of the components and ports that are relevant to the given slicing criterion.

The main components of the DSAS method are *Event Filter* (D.2) and *Forward Architecture Slicer* (D.3).

Whenever *Event Filter* gets an event information, ADL specific events are filtered out to prevent *Forward Architecture Slicer* from confusion -- e.g., “START” events from RAPIDE are filtered out since these events are specific to RAPIDE to inform that components are instantiated (or *started*) and they are not related to the architecture slice.

3.2 The DSAS Algorithm

The algorithm computes architecture slices in a forward manner, which means that it computes slices as the target program executes. During program execution, the algorithm maintains a set of components and connectors where the current event is received or transmitted.

The DSAS algorithm is shown in Figure 3, and functions and data structure used in the algorithm are as follows:

- ◇ $e_cntr = \#(e_{sc})$ is a counter for number of occurrences of e_{sc} , where $\#:E \rightarrow \mathbf{N}$.
- ◇ $\text{SliceSet} \subseteq 2^P$.
- ◇ $\text{SliceSubset}, S_k \subseteq P$, is a subset of slice such that $S_k \in \text{SliceSet}$, where $e_x \in S_k$ and $e_x \in P = E$. S_k 's are created when there is a concurrency, and events are added to the proper S_k 's according to the causality.
- ◇ $ns = |\text{SliceSet}|$, the cardinality of SliceSet .
- ◇ *CausalPredecessor*: $E \rightarrow E$ is a function where $\text{CausalPredecessor}(e_x) = e_y$ such that $e_x \in E$ is reachable from $e_y \in E$ and $\forall e_j \in E [((\text{CausalPredecessor}(e_j) = e_y) \wedge (\text{CausalPredecessor}(e_x) = e_j)) \rightarrow ((e_j=e_y) \vee (e_j=e_x))]$ i.e., e_x is immediately reachable from e_y .

Input:
software architecture, A
slicing criterion, $C^d = (I_{sc}, e_{sc}, n_{sc})$

Output:
software architecture slice, S_A

Algorithm_DSAS: $A \times C^d \rightarrow S_A$

begin algorithm

- 1: $e_cntr := 0; ns := 0; SliceSet := \emptyset;$
- 2: **while** $(\exists e \wedge (e_cntr < n_sc))$
- 3: **if** $(ns = 0)$ **then**
- 4: $S_{ns} := \{e\};$
- 5: $ns := ns + 1;$
- 6: **elseif** $(\exists \text{concurrency})$ **then**
- 7: $e_concurrent := \{e_p \mid \text{concurrency generates } k \text{ concurrent events, } e_m, e_{m+1}, \dots, e_{m+k-1}, \text{ and } p = m, m+1, \dots, (m+k-1)\};$
- 8: **foreach** $ec \in e_concurrent$ **do**
- 9: $S_{ns} := \{ec\};$
- 10: $ns := ns + 1;$
- 11: **end do**
- 12: **else**
- 13: **if** $(\forall i \wedge \exists e \in S_i)$ **then** /* event loop */
- 14: $ep := e;$
- 15: **do**
- 16: **if** $(e \neq \text{CausalPredecessor}(ep))$ **then**
- 17: $*\text{CausalPredSet}(ep) :=$
- 18: $\text{CausalPredSet}(ep) \setminus$
- 19: $\{\text{CausalPredecessor}(ep)\};$
- 20: **else**
- 21: $*\text{CausalPredSet}(ep) :=$
- 22: $\text{CausalPredSet}(ep) \cup \{e\};$
- 23: **fi**
- 24: $ep := \text{CausalPredecessor}(ep);$
- 25: **while** $(ep \neq e)$
- 26: **else**
- 27: $*\text{CausalPredSet}(e) :=$
- 28: $\text{CausalPredSet}(e) \cup \{e\};$
- 29: **fi**
- 30: **fi**
- 31: **if** $(e = e_{sc})$ **then** $e_cntr := e_cntr + 1;$ **fi**
- 32: /* Recompute architecture slice according to the dependency */
- 33: $tcs := \text{CausalPredSet}(e);$
- 34: $S_A := tcs;$
- 35: **while** $(\text{CausalPredSet}(\text{First}(tcs)) \neq \emptyset)$
- 36: $tcs := \text{CausalPredSet}(\text{First}(tcs));$
- 37: $S_A := S_A \cup tcs;$
- 38: **end while**
- 39: **end while**
- 40: **print** $S_A;$
- 41: **end algorithm**

Figure 3 DSAS algorithm

- ◇ $\text{CausalPredSet}: E \rightarrow \text{SliceSubset}$ is a function where $\text{CausalPredSet}(e_x) = S_k$ such that $e_y \in S_k$ and the two events $e_x, e_y \in E$ have relationship of $\text{CausalPredecessor}(e_x) = e_y$. Note that $*\text{CausalPredSet}(e_x)$ is a set itself that contains e_y as an element, instead of a return value from the function.
- ◇ $\text{First}: \text{SliceSubset} \rightarrow E$ is a function such that $\text{First}(S_k) = e_x, e_x \in S_k$ if $\forall e_y \in S_k [e_x \Rightarrow e_y]$. That is, e_x is the causal predecessor of all $e_y \in S_k$.
- ◇ $tcs \in \text{SliceSet}$ is a temporary set.
- ◇ $ec, ep \in E$ are temporary events.
- ◇ $e \in E$ is an event input from the executable.

The DSAS algorithm reduces the length of the event trace whenever it encounters an *event loop* as depicted in Figure 4 (a).

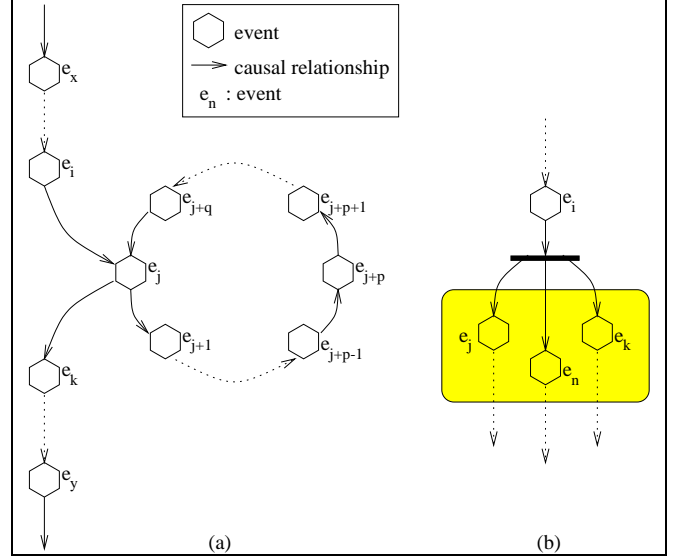


Figure 4 Loop and concurrency

There are two different cases of handling event loops: (i) e_{sc} is an event loop or (ii) e_{sc} is not in the event loop. For the first case, assume that $e_{sc} = e_{j+p}$ in Figure 4 (a), i.e. e_{sc} is a part of an event loop. In this case, the software architecture slice should include the event path $e_x, \dots, e_i, e_j, e_{j+1}, \dots, e_{j+p}$.

The architecture slice for the second case is much different from the first one. We assume that $e_{sc} = e_y$. In this case the event sequence of $e_j, e_{j+1}, \dots, e_{j+q}$ may occur several times and some of the variables which affect the path outside the event loop could be changed --- e.g., variables(parameters) which are affected inside a loop may participate in part as *guards* outside of a loop later. But in the sense of software architecture, event loop can be considered as an event where loop starts and ends, so that the event loop can be safely removed from the slice.

In other words, when multiple events happen at a port, there exists a cycle of events that can be removed from the current set as the events involved during the cycle either do not affect the occurrence of e_{sc} or re-occur later.

For this reason, the slice contains only $e_x, \dots, e_i, e_j, e_k, \dots, e_y$ as a part of slice. The lines 12--19 of DSAS algorithm detects an event loop and deletes the *ports* which are involved in the event loop.

The other issue is the handling of *concurrency* as shown in Figure 4 (b). By virtue of concurrency, all the paths of concurrent events should be maintained in a parallel manner.

Whenever a concurrency is encountered, the DSAS algorithm (lines 7--10) creates new separate subsets for each concurrent events.

All the other events are inserted into an appropriate slice subset (S_i) which contain the causal predecessor event of the current event, using the *CausalPredSet* function (line 21 in the algorithm).

4 Slicing EOPS

Returning to the system under study shown earlier in Figure 1, we now show how to compute an architecture slice for *EOPS*.

For the purpose of showing the intermediate and final results, we use RAPIDE[14] and one of its tools, called *pov*³. *pov* has the capability of displaying the event trace of the software architecture graphically with such display options as forward or backward events selection and event ordering by causality, start-time, end-time, etc.

Figure 2 shows the relationships among the phases of the DSAS methodology and the various diagrams. The computation of the *forward dynamic architecture slice* of EOPS with respect to the *slicing criterion* can be described as follows:

- (A) *Software architecture* of EOPS can be described by a diagram that consists of *components* and *connectors* between the *ports* of the *components* as shown in Figure 1.
- (B) Implementation of EOPS using ADLs. After phase (A), the architecture is implemented using some ADL of choice. In this paper, we use RAPIDE as an ADL and parts of the implemented code is shown in Figure 9 in Appendix. The implemented architecture is then compiled to make an executable.
- (C) *Slicing Criterion*. As mentioned earlier in Section 2.3, the *slicing criterion* consists of initial values of the variables, the event name of interest and its number of occurrences. In our example, the event name is a port name, as mentioned in Section 4. In the slicing criterion, we assume that we are interested in the second occurrence (2) of an item event (P1) in an Accounting component (C7) as shown in Figure 6. The *initial values* for customer id is 84, ordered item numbers are 9, 6, 1 and number of items for each are 2, 1, 5, and the event name is C7:P1⁴ and its occurrence count is 2. Then the slicing criterion becomes $C_A^d = (\{ \langle \text{CustId}, 84 \rangle, \langle \text{Item}_1, 9 \rangle, \langle \text{N}_1, 2 \rangle, \langle \text{Item}_2, 6 \rangle, \langle \text{N}_2, 1 \rangle, \langle \text{Item}_3, 1 \rangle, \langle \text{N}_3, 5 \rangle, \text{C7:P1}, 2 \})$

- (D) Running *Forward Dynamic Architecture Slicer*. *Forward Dynamic Slicer* gets an EOPS architecture

and slicing criterion as input. *Forward Architecture Slicer* builds a table similar to Table 1 in the Appendix.

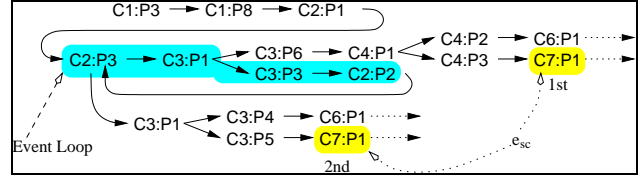


Figure 5 Event sequence from EOPS

Events	Architecture Slice
C1:P3	C1:P3
C1:P8	C1:P3, C1:P8
C2:P1	C1:P3, C1:P8, C2:P1
C2:P3	C1:P3, C1:P8, C2:P1, C2:P3
C3:P1	sf C1:P3, C1:P8, C2:P1, C2:P3, C3:P1
C3:P3, C3:P6	C1:P3, C1:P8, C2:P1, C2:P3, C3:P1, C3:P3, C3:P6
C4:P1	C1:P3, C1:P8, C2:P1, C2:P3, C3:P1, C3:P3, C3:P6, C4:P1
C2:P2	C1:P3, C1:P8, C2:P1, C2:P3, C3:P1, C3:P3, C3:P6, C4:P1, C2:P2
C2:P3	C1:P3, C1:P8, C2:P1, C2:P3
...	...
C3:P1	C1:P3, C1:P8, C2:P1, C2:P3, C3:P1
C3:P4, C3:P5	C1:P3, C1:P8, C2:P1, C2:P3, C3:P1, C3:P4, C3:P5
C6:P1, C7:P1	C1:P3, C1:P8, C2:P1, C2:P3, C3:P1, C3:P4, C3:P5, C6:P1, C7:P1
-	C1:P3, C1:P8, C2:P1, C2:P3, C3:P1, C3:P5, C7:P1
...	C1:P3, C1:P8, C2:P1, C2:P3, C3:P1, C3:P5, C7:P1

Table 1 Computation of architecture slice for EOPS

- (D.1) The ADL executable of EOPS generates a sequence of events. A part of the trace is shown in Figure 1. It shows two *event property* windows, one for ship and the other for done. These windows show the *source* and the *destination* components of the events along with parameters. For example, the event done is originated from Order_Entry'41 and becomes CGI_payment_req at the receiving component. A parameter associated with this event is the customer number 84. Note that the *start time* and the *end time* are all "0s" because *time* (or *delay*) is not involved in this case. Several "START" events shown in Figure 9

³ *pov* is one of the tools built by the members of the RAPIDE project.

⁴ We rename the components and connectors for the sake of simplicity as shown in Table 3 in Appendix. For example, C2:P4 is an *import/outport* 4 of *component* 2.

indicates that the same number of components in the architecture are started during the execution of the ADL executable.

- (D.2) *Event Filter* receives the events from the ADL executable and filters out the ADL specific events such as “START”.
- (D.3) *Forward Architecture Slicer* gets an EOPS architecture and the filtered events from *Event Filter* as input. For each occurrence of a filtered event, the *Forward Architecture Slicer* computes corresponding architecture slices. The ADL executable of EOPS generates a sequence of events as shown in Figure 5 according to the given input. The shaded area in Figure 5 shows the presence of an *event loop*. The architecture slice is computed by the DSAS algorithm. For example, when *Forward Architecture Slicer* receives a filtered event C2:P3, the architecture slice becomes $\{C1:P3, C1:P8, C2:P1\} \cup \{C2:P3\} = \{C1:P3, C1:P8, C2:P1, C2:P3\}$.

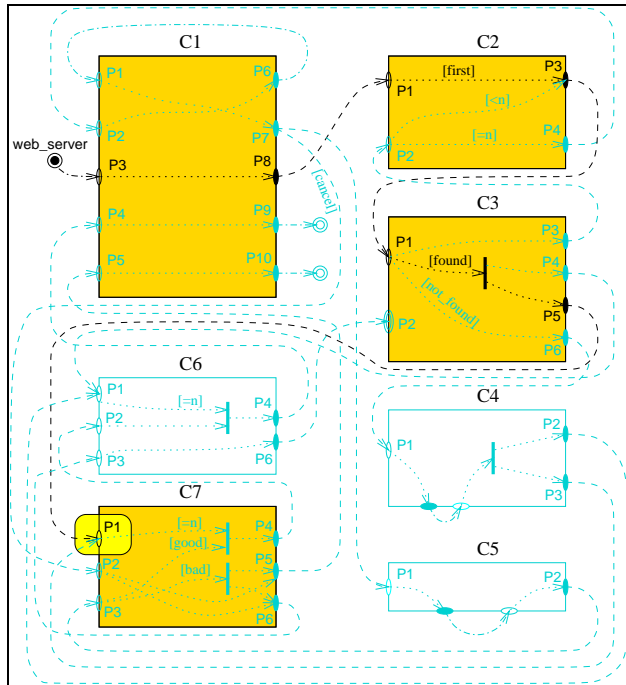


Figure 6 Dynamic architecture slice of EOPS

When the DSAS algorithm receives the event C2:P3, which creates an event loop, all the events that are involved in the loop so far are removed from the architecture slice and the resulting slice becomes $\{C1:P3, C1:P8, C2:P1, C2:P3, C3:P1, C3:P3, C3:P6, C4:P1, C2:P2\} \setminus \{C3:P1, C3:P3, C3:P6, C4:P1, C2:P2\} = \{C1:P3, C1:P8, C2:P1, C2:P3\}$.

- (E) *Final Architecture Slice*. When the given *slicing criterion* is satisfied, *Forward Dynamic Slicer* stops and prints the resulting architecture slice. The resulting architecture slice contains only the *components* and *ports* that are relevant to the slicing criterion. The resulting architecture slice is $S_A = \{C1:P3, C1:P8, C2:P1, C2:P3, C3:P1, C3:P5, C7:P1\}$. The graphical representation of the result is shown in Figure 6. The shaded portion is the resulting architecture slice. For the purpose of comparing the original architecture and its slice, partial RAPIDE code and its resultant architecture slice code for EOPS are shown in Figure 7 and Figure 8 in Appendix.

The event traces of EOPS and its slice are shown in Figure 9 and Figure 10 respectively. We can see significant reduction of the architecture in Figure 10.

5 Related Works

Our contribution includes the proposal of the notion of dynamic software architecture slicing or DSAS, some extension of ADL with some features of Petri-Net to resolve possible discrepancies in design and the use of RAPIDE tools to support our approach. We also presented a methodology for DSAS and its algorithm. Our *Forward Dynamic Software Architecture Slicer* is shown in a shaded cell along with the other related works in Table 2.

Type	Static	Dynamic
Program	[Weiser84][Ottenstein84][Horwitz90]	[Agrawal90][Korel86][Korel94][Song99]
Software Architecture	[Stafford98][Zhao97]	Forward Dynamic Architecture Slicing

Table 2 Categories of related work

The concept of program slicing was first proposed by Weiser [19], and later extended by Agrawal, Hogan [1], Korel and Laski [9] to dynamic program slicing so as to identify unique dynamic behavior of a program under a given input with respect to some variable of interest. Korel and Yalamanchili [10] proposed a way of computing program slices namely, *forward dynamic program slicing* that doesn't require the construction of dependency graphs to compute program slices. There have been some works on the statement level forward computation of program slices [17]. A statement level slice consists of actual lines of code that is subset of the original program.

In this paper we focus on software architecture slicing while employing architectural description languages or ADLs. Among the currently available ADLs such as ACME [6], Aesop [5], Adage [4], Meta-H [3], C2 [12], RAPIDE [11, 14], SADL [13], UniCon [15], and Wright [2], we are interested in event-driven ADLs, in particular RAPIDE as it offers a tool for prototyping software

architectures. Computations are defined in an event pattern language. Patterns are sets of events together with their partial ordering, represented by a so-called *poset*, which we use for the purpose of dynamic analysis.

Zhao introduced the concept of static software architecture slicing [20]. His approach uses ACME and Wright and constructs software architectural dependence graph to compute architecture slice. The result is an architecture slice that isolates the behavior of a specified set of component's ports or a connector's roles. Stafford *et al.* [16] investigated architectural level dependency analysis technique called chaining and implemented the technique in a tool called Aladdin. Aladdin takes as input an architectural specification and produces chains representing dependence relationships.

Kim *et al.* [7] introduced the notion of dynamic software architecture slicing which reveals the dynamic behavior of the architecture through the precise dependency among the set of components and connectors that are relevant to the particular variable-value assignments and events as expressed in the slicing criterion. In this paper, we firstly offer a cleaner notation for the specification of software architecture and slicing criterion. Secondly, the (graphical) notation of ADLs mentioned above has been extended with the *concurrency* features of the Petri-Net formalism in order to describe the concurrent, dynamic behavior of a software architecture, and consequently the algorithm has been adapted to handle concurrency. Lastly, we use an *e-commerce* example instead of an operating system example. For further extensions, we are investigating other constructs such as *timing constraints*.

6 Conclusion

Albeit the pivotal role that software architecture is expected to play in software engineering, understanding, modeling and analyzing such a conceptual architecture tends to be non-trivial due to its abstract nature. Towards in-depth understanding of such a high level of abstraction, we have introduced the notion of *dynamic software architecture slicing* (DSAS), along with a methodology to support its use and an algorithm to generate dynamic architecture slices. Through a small study with the use of an event tracing tool, we have demonstrated the feasibility and expected benefits of our approach.

A dynamic software architecture slice represents the particular sequence of parts of the components and connectors of the given architecture during the (concurrent) execution of one of its corresponding program with respect to a particular software architecture slicing criteria. The notion of DSAS goes beyond the notion of static software architecture slicing while exploiting the benefits of the "dynamic" aspect of dynamic program slicing, namely, the capability to

generate potentially a much smaller number of components and connectors in each slice.

Being the first in its kind, however, there are clearly several avenues of improvement for our work. A formalization is needed of the notion of DSAS, in particular the precise relationship between a dynamic architecture slice and its corresponding program slices. Another line of future work concerns more powerful slicing criterion so as to enhance the utility of DSAS. Work is underway to enrich the current criteria, consisting of a set of events and variable-value bindings, towards fault injection and detection capabilities. The tracing tool also needs to be extended with the dynamic slicing algorithm so as to produce software architecture slices instead of program slices. With these improvements, we feel that the DSAS approach can be a basis towards the systematic understanding, modeling and analysis of software architecture.

7 Reference

- [1] H. Agrawal and J. R. Horgan, "Dynamic Program Slicing", *Proc. ACM SIGPLAN'90*, 1990, pp. 246-256.
- [2] R. Allen and D. Garlan, "Formalizing Architectural Connection", *Proc. 16th International Conference on Software Engineering*, May 1994, pp. 71-80.
- [3] P. Binns and S. Vestal, "Formal Real-time architecture Specification and Analysis", *Tenth IEEE Workshop on Real-time Operating Systems and Software*, May 1993.
- [4] L. Colianese and R. Szymanski, "DSSA-ADAGE: An Environment for Architecture-based Avionics Development", *Proc. AGARD'93*, May 1993.
- [5] D. Garlan, R. Allen and J. Ockerbloom, "Exploiting Style in Architectural Design Environments", *Proc. ACM SIGSOFT*, Dec. 1994, pp. 179-185.
- [6] D. Garlan, R. Monroe and D. Wile, "Acme: An Architecture Description Interchange Language", *Proc. CASCON '97*, 1997.
- [7] T. Kim, Y. Song, L. Chung and D. T. Huynh, "Dynamic Software Architecture Slicing", *COMPSAC 99*, Phoenix, Arizona, 1999.
- [8] T. Kim, Y. Song, L. Chung and D. T. Huynh, "Software Architecture Analysis Using Dynamic Slicing", *AoM/IAoM 17th International Conference on Computer Science*, San Diego, California, 1999.
- [9] B. Korel and J. Laski, "Dynamic Slicing of Computer Programs", *Journal of Systems Software*, 1990, pp. 187--195.

- [10] B. Korel and S. Yalamanchili, "Forward Computation of Dynamic Program Slices", *ISSTA 94*, Seattle Washington, 1994, pp. 66--79.
- [11] D. C. Luckham, J. J. Kenney and L. M. Augustin, "Specification and Analysis of System Architecture Using Rapide", *IEEE Trans. Software Eng.*, vol. 21, no. 4, April 1995, pp. 336-354.
- [12] N. Medvidovic, P. Oreizy, J. E. Robbins and R. N. Taylor, "Using Object-Oriented Typing to Support Architectural Design in C2 Style", *SIGSOFT'96: Proc. Fourth ACM Symposium on the Foundations of Software Engineering*, October 1996.
- [13] M. Moriconi, X. Qian and R. Riemenschneider, "Correct Architecture Refinement", *IEEE Trans. on Software Engineering, Special issue on Software Architecture*, vol. 21, no. 4, April 1995, pp. 356-372.
- [14] PAVG, Computer Science Lab, "Guide to the Rapide 1.0 Language Reference Manuals", Stanford University, July, 1997.
- [15] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Trans. Software Engineering, Special issue on Software Architecture*, vol. 21, no. 4, April 1995, pp. 314-335.
- [16] J. A. Stafford, D. J. Richardson and A. L. Wolf, "Aladdin: A Tool for Architecture-Level Dependence Analysis of Software Systems", *Technical Report CU-CS-858-98*, University of Colorado, Department of Computer Science, April 1998.
- [17] Y. Song and D. T. Huynh, "Forward Dynamic Object-Oriented Program Slicing", *Proc. Application Specific Systems and Software Engineering '99*, March 1999.
- [18] J. M. Tenenbaum, "WISs and Electronic Commerce", *CACM*, vol. 41, no. 7, July 1998.
- [19] M. Weiser, "Program Slicing", *IEEE Trans. on Software Engineering*, vol. 10, No. 4, July 1984, pp. 352--357.
- [20] J. Zhao, "Slicing Software Architectures", *Technical Report 97-SE-117*, Information Processing Society of Japan, November 1997, pp. 85--92.

8 Appendix

8.1 Naming Convention

Table 3 shows the abbreviated notation used in the EOPS example.

Component	Sym	Port	Sym
Order_Req_Handler	C1	CGI_payment_info	C1:P1
		payment_req	C1:P2
		CGI_order	C1:P3
		order_success	C1:P4
		order_fail	C1:P5
		CGI_payment_req	C1:P6
		credit_info	C1:P7
		place_order	C1:P8
		CGI_ship_info	C1:P9
		CGI_order_rej	C1:P10
Order_Entr	C2	take_order	C2:P1
		next_item	C2:P2
		ship_item	C2:P3
		done	C2:P4
Inventory	C3	find_item	C3:P1
		restock_items	C3:P2
		get_next	C3:P3
		ship	C3:P4
		add_item	C3:P5
Back_Order	C4	take_back_order	C4:P1
		ship	C4:P2
		add_item	C4:P3
Credit_Checker	C5	check_req	C5:P1
		credit_res	C5:P2
Shipping	C6	recv_item	C6:P1
		recv_receipt	C6:P2
		cancel	C6:P3
		shipping_info	C6:P4
		restock	C6:P5
Accounting	C7	items	C7:P1
		cancel	C7:P2
		credit_res	C7:P3
		issue_receipt	C7:P4
		fail	C7:P5
		restock	C7:P6

Table 3 Name simplification in EOPS

8.2 Example RAPIDE Partial Code

Parts of the RAPIDE code has been shown here. Note that the lines start with “//” become inactive (or commented out) in the Figure 8.

```

.....
type Order_Req_Handler is interface
action
  in  CGI_payment_info(C : CustId;
                      CCN : CreditCardNo),
      Payment_Req(C : CustId),
      CGI_Order(C : CustId; OL : Item_List),
      Order_Success(C : CustId),
      Order_Fail(C : CustId);
  out CGI_Payment_Req(C : CustId),
      Credit_Info(C : CustId;
                  CCN : CreditCardNo),
      Place_Order(C : CustId; OL : Item_List),
      CGI_Ship_Info(C : CustId),
      CGI_Order_Rej(C : CustId);
behavior
begin
  (?C : CustId; ?CCN : CreditCardNo)
  CGI_payment_info(?C, ?CCN)
  ||> Credit_Info(?C, ?CCN); ;
  (?C : CustId)
  Payment_Req(?C)
  ||> CGI_Payment_Req(?C); ;
  (?C : CustId; ?IL : Item_List)
  CGI_Order(?C, ?IL)
  ||> Place_Order(?C, ?IL); ;
  (?C : CustId)
  Order_Success(?C)
  ||> CGI_Ship_Info(?C); ;
  (?C : CustId)
  Order_Fail(?C)
  CGI_Order_Rej(?C); ;
end Order_Req_Handler;
.....
architecture ecomm01() return root
is
  ORQH : Order_Req_Handler;
  OENT : Order_Entry;
  INVT : Inventory;
  BORD : Back_Order;
  CRDC : Credit_Checker;
  .....
connect
  -- Order_Req_Handler
  (?C : CustId; ?OL : Item_List)
  ORQH.Place_Order(?C, ?OL) to
    OENT.Take_Order(?C, ?OL);
  (?C : CustId)
  ORQH.CGI_Payment_Req(?C) to
    WCGI.CGI_Payment_Req(?C);
  (?C : CustId; ?CCN : CreditCardNo)
  ORQH.Credit_Info(?C, ?CCN) to
    CRDC.Check_Req(?C, ?CCN);
  .....
  -- Order_Entry
  (?C : CustId; ?I : ItemNo; ?N : Quantity)
  OENT.Ship_Item(?C, ?I, ?N) to
    INVT.Find_Item(?C, ?I, ?N);
  (?C : CustId)
  OENT.Done(?C) to
    ORQH.Payment_Req(?C);
  .....
end ecomm01;

```

Figure 7 Initial Rapide code of EOPS

```

.....
type Order_Req_Handler is interface
action
  in  // CGI_payment_info(C : CustId;
                          // CCN : CreditCardNo),
      // Payment_Req(C : CustId),
      CGI_Order(C : CustId; OL : Item_List);
      // Order_Success(C : CustId),
      // Order_Fail(C : CustId);
  out // CGI_Payment_Req(C : CustId),
      // Credit_Info(C : CustId;
                    // CCN : CreditCardNo),
      Place_Order(C : CustId; OL : Item_List);
      // CGI_Ship_Info(C : CustId),
      // CGI_Order_Rej(C : CustId);
behavior
begin
  // (?C : CustId; ?CCN : CreditCardNo)
  // CGI_payment_info(?C, ?CCN)
  // ||> Credit_Info(?C, ?CCN); ;
  // (?C : CustId)
  // Payment_Req(?C)
  // ||> CGI_Payment_Req(?C); ;
  // (?C : CustId; ?IL : Item_List)
  // CGI_Order(?C, ?IL)
  // ||> Place_Order(?C, ?IL); ;
  // (?C : CustId)
  // Order_Success(?C)
  // ||> CGI_Ship_Info(?C); ;
  // (?C : CustId)
  // Order_Fail(?C)
  // CGI_Order_Rej(?C); ;
end Order_Req_Handler;
.....
architecture ecomm01Slice() return root
is
  ORQH : Order_Req_Handler;
  OENT : Order_Entry;
  INVT : Inventory;
  // BORD : Back_Order;
  // CRDC : Credit_Checker;
  .....
connect
  -- Order_Req_Handler
  (?C : CustId; ?OL : Item_List)
  ORQH.Place_Order(?C, ?OL) to
    OENT.Take_Order(?C, ?OL);
  // (?C : CustId)
  // ORQH.CGI_Payment_Req(?C) to
  // WCGI.CGI_Payment_Req(?C);
  // (?C : CustId; ?CCN : CreditCardNo)
  // ORQH.Credit_Info(?C, ?CCN) to
  // CRDC.Check_Req(?C, ?CCN);
  .....
  -- Order_Entry
  (?C : CustId; ?I : ItemNo; ?N : Quantity)
  OENT.Ship_Item(?C, ?I, ?N) to
    INVT.Find_Item(?C, ?I, ?N);
  // (?C : CustId)
  // OENT.Done(?C) to
  // ORQH.Payment_Req(?C);
  .....
end ecomm01Slice;

```

Figure 8 Rapide code slice of EOPS

8.3 RAPIDE Event Trace of EOPS

The screen dump (drawn by *pov*) of the partial event trace for EOPS with five orders is shown in Figure 9. Figure 10 shows the event trace of the resulting EOPS architecture slice.

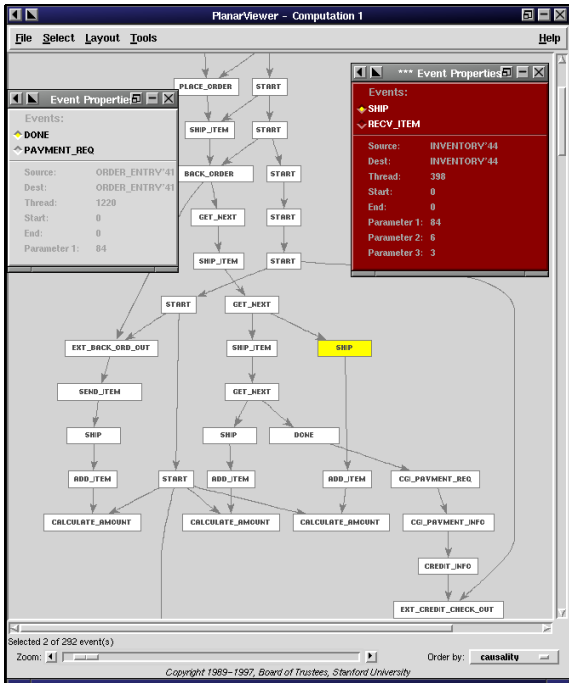


Figure 9 The Rapide event trace of EOPS (partial view)

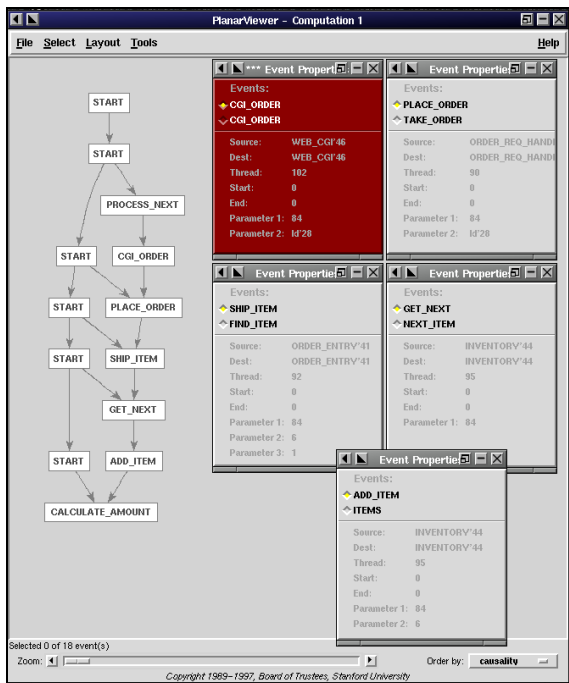


Figure 10 The Rapide event trace per slicing criterion



Taeho Kim received the MS degree in 1992, is currently working toward a Ph.D. degree, in computer science from the University of Texas at Dallas. He is also with Alcatel Corporate Research Center, Richardson, TX. His research interests include software architecture, e-commerce, computer network architectures and their protocol verifications.



Dr. Song joined UALR in 1999 as an assistant professor. He has received his master's and doctoral degrees on computer science from the University of Texas at Dallas at 1992 and 1999, respectively. He started his career as a software developer back in mid 80's. Since then he has about 9 years industry experience as a software engineer before he joins UALR. His research interests are program slicing, program analysis, software architecture, e-commerce, object-orientation, program comprehension, software testing, and software engineering.



Dr. Lawrence Chung joined UTD in 1994, and currently is an Assistant Professor of Computer Science in the Erik Jonsson School of Engineering and Computer Science. He received his Ph.D. in Computer Science in 1993 from the University of Toronto, where he had previously received the B.Sc. and M.Sc. degrees. His research interests include Requirements Engineering and Software Architecture, as well as Electronic Commerce/Business Architecting. He has recently coauthored a book, "Non-Functional Requirements in Software Engineering", which is being adopted in extending object-oriented analysis to goal-oriented analysis.



Professor Huynh received the M.S. and Ph.D. degrees in Computer Science from the University of Saarlandes (Germany) in 1977 and 1978, respectively, where he remained as a postdoctoral research assistant until 1982. From 1982 to 1983 he was a Visiting Assistant Professor at the University of Chicago. He then spent three years as an Assistant Professor of Computer Science at Iowa State University before joining the Computer Science faculty at the University of Texas at Dallas as an Associate Professor in 1986. Dr. Huynh was promoted to Full Professor in 1991,

and became Program Head of the Computer Science Program in May 1997. He has been a member of the Advisory Board of the Journal of Automata, Languages and Combinatorics since 1996. His research interests include Computational complexity theory, automata and formal languages, concurrency theory, communications networks and protocols, parallel computation, and software metrics.