Dynamic Software Architecture Slicing

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

Abstract

Software architectural design is becoming increasingly important in software engineering, as being manifested through various recent developments in the field such as the component-based software engineering paradigm and the distributed and collaborative computing paradigm. Abstraction is such a mechanism as the key concept underpinning software architecture, namely hiding the immense amount of details. Despite its long-recognized benefits, however, abstraction can also pose difficulties with the understanding and analysis of software architecture since one architecture can result in potentially an infinite number of different system behaviors. In order to alleviate such difficulties, we introduce the notion of dynamic software architecture slicing *(*DSAS*), a methodology for using the notion, and an algorithm to generate dynamic software architecture slice. We demonstrate the feasibility and the expected benefits of the approach by using an illustrative example.*

Keywords*: software architecture, dynamic slicing.*

1 Introduction

Software architectural design is becoming increasingly important in software engineering, as being manifested through various recent developments in the field such as the component-based software engineering paradigm and the distributed and collaborative computing paradigm. This increasing importance is largely due to the need for a mechanism which would enable the software developer to deal with the ever-expanding complexity and size of software systems.

Abstraction is such a mechanism as the key concept underpinning software architecture, namely hiding the immense amount of details about the data structures, algorithms, variations in programming language constructs, etc. which would all be needed eventually for the implementation of the projected system. As the highest level of abstraction, a software architecture defines a software system in terms of components which carry out computations, connectors which are used by the components to interact with each other, constraints that are imposed on the behavior of components and connectors, etc. [1]. Despite its long-recognized benefits, however, abstraction can also pose difficulties with the understanding and analysis of software architecture since one architecture can result in potentially an infinite number of different system behaviors.

In order to alleviate such difficulties, we introduce the notion of *dynamic software architecture slicing* (*DSAS*), in which 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 (e.g., a set of resources and their values) provided by the software architect.

The notion of DSAS utilizes, and extends, a couple of different types of slicing techniques, one from program slicing and the other from static software architecture slicing.

Type / Level	Static	Dynamic
Program	[Weiser84][Ottenstein84] [Horwitz90]	[Agrawal90] [Korel86] [Korel94] [Song98][Song99]
Software Architecture	[Stafford98][Zhao97]	Forward Dynamic Architectural Slicing

Table 1. Categories of related work

While a program slice is the set of program statements that are relevant to the particular variables of interest at some point during the program level execution, a software architectural slice is the set of (parts of) architectural components and connectors that are relevant to the particular variables and events of interest at some point during the architectural level execution. For both types, a static slice is determined independently of the input at compile time typically through the dependency analysis technique. A dynamic slice, on the other hand, is determined according to the input at run time, hence smaller in size than its static counterpart. As such, a dynamic slice helps to isolate a par-

ticular execution path.

The distinctives of our work include: the proposal of the notion of dynamic software architecture slicing or DSAS, some extension of ADL with some features of Petri-Net in order to enhance conceptual expressive power. We also presented the 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 1 [2] [3] [4] [5] [6] [7].

The focus of this paper is dynamic software architecture slicing, namely, revealing 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.

2 Dynamic Software Architecture 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

Typically a *printer subsystem* is a part of an operating system (e.g., the UNIX operating system) which controls and manipulates various print requests for the sharing of printers among multiple users. Suppose, for example, that a user wants to print a report and issues a print request, i.e., the print command together with the name of the file, the name of the printer, the number of copies to be printed, single-/double-sided, etc. The operating system accepts the request and fulfills the print request by the use of various processes, which can involve a number of decision points and cycles.

The software architecture consists of seven components and a number of connectors between the components. All the seven components of the printer subsystem are daemons, i.e., processes that are active all the time: one user shell (User); one printer queue daemon (Queue); one line printer daemon (LPD); two filter daemons, PostScript filter daemon (PS_Flt_D) and ASCII filter daemon (ASC_Flt_D); and two printer daemons, ASCII printer daemon (ASC_Prt_D) and PostScript printer daemon (PS_Prt_D).

The first component, User, accepts the print command from the user through the start port as an event (start), which interprets user commands in terms of the internal user id (u) and the print job (j) — e.g., PostScript or ASCII file, and generates a send_job event at the send_job port. As can be seen here, our convention is that the name of an event is identical with that of the event where the event is received

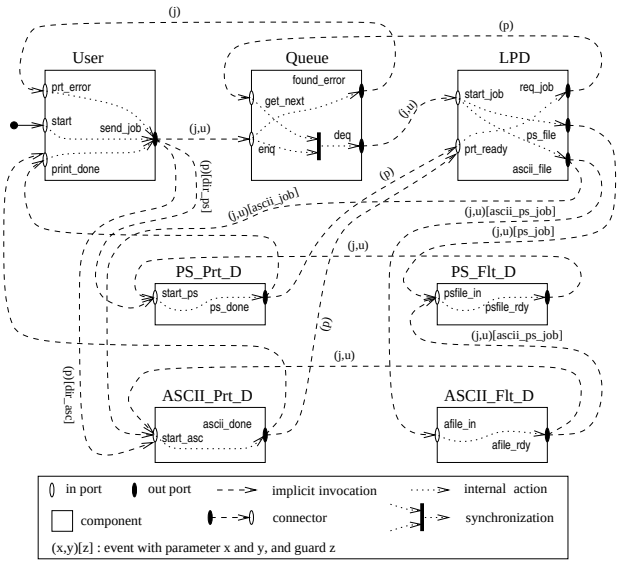

Figure 1. Printer subsystem architecture diagram

or transmitted. The underlying event handler of the operating system then detects this event and notifies (or triggers) either the printer queue daemon or one of the two types of printer daemons.

2.2 Basic Definitions

Now we define the basic concepts of *software architecture* and *architecture slice*.

Definition 1 *A software architecture can be defined as*

$$\mathcal{A} = (C, P, \Delta, \Gamma)$$

where

- C *is a finite set of* components.
- P *is a finite set of* ports *(*events*) and* $P = P_i \cup P_o$ *where* P_i *and* P_o *are finite sets of* inports *and* outports*, respectively. The set of* inports *of a component* C *is denoted by* P_i^C *and the set of* outports *of a* C *is denoted by* P_o^C*. An* event *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 is associated with a* guard *which is a Boolean expression and a set of parameters.*
- Γ *is a finite set of* internal guards *or* actions *of* C*, written as* $P_i \xrightarrow{\gamma} P_o$*. We use* $P_i \xrightarrow{\gamma} P_o$ *to refer to the path from* P_i *to* P_o *through* $\gamma \in \Gamma$.

Internal path is $\Psi = \{p_i \xrightarrow{\gamma} p_o \mid p_i \in P_i,\ p_o \in P_o,\ \gamma \in \Gamma\}$. And a part of a component is a member of $\Psi \cup P_i^C \cup P_o^C$.

Definition 2 *Software architecture slice can be defined as*

$$\mathcal{S}_{\mathcal{A}} = (C', P', \Delta')$$

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, $\Delta' \subseteq \Delta$.

2.3 Software Architecture Slicing Criterion

In software architecture slicing, we are interested in *components* and *connectors*. When dynamic software architecture slicing is considered, occurrences of the events are of main concern since we are interested in the causality of events which drives the behavioral characteristics of the architecture.

Now we define dynamic software architecture slicing criterion as follows.

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

$$\begin{aligned} \mathcal{C}_{\mathcal{A}}^{d} &= (init\_val\_set,\ comp : port,\ event\_num) \\ &= (I, c_c : [p_i^{c_c} | p_o^{c_c}],\ n) \end{aligned}$$

where I *is a finite set of initial values,* $comp : port$ *is an event to be observed, and* $event\_num$ *is an event sequence number. Note that* c_c *denotes the component in the slicing criterion and* $p_i^{c_c}$ *and* $p_o^{c_c}$ *denote* inport *and* outport *of* c_c*, respectively.*

3 The DSAS Methodology and Algorithm

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

3.1 The DSAS Methodology

The computation of *forward dynamic software architecture slice* is shown in Figure 2. The description of DSAS methodology is as follows:

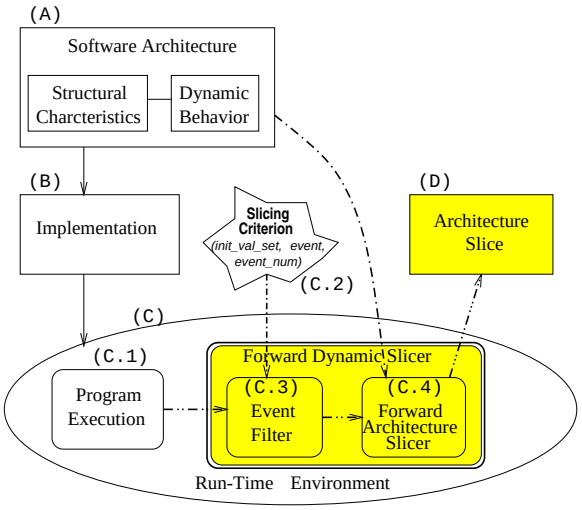

Figure 2. Dynamic software architecture slicing methodology

(A) *Software Architecture*. Software Architects together with requirements engineers design *Software Architecture*. It reflects the *requirements* of the software system. Software 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 using ADLs*. After phase (A), the architecture is implemented using some ADL of choice. A software architecture can be implemented at architectural level by using some of the popular architectural description languages (ADLs) such as ACME [8], RAPIDE [9], Aesop [10], UniCon [11], and Wright [12]. After the software architecture is implemented by an ADL of choice, it then be compiled to make an executable. Among those ADLs, we are focusing on event-driven ADLs such as ACME and RAPIDE.

(C) *Run-time*. During the run-time, *Forward Dynamic Slicer* executes the executable and reads its ADL to identify component and connector information along with the event names used in the ADL.

- (C.1) *Program Execution*. The ADL executable generates a *bag of events*. The *Forward Dynamic Slicer* gets the event and its sequence number of interest in the form of slicing criterion.
- (C.2) *Slicing Criterion*. In the slicing criterion, the initial condition (a finite set of $variable : value$) is also given.
- (C.3) *Event Filter*. When *Forward Dynamic Slicer* receives events from the ADL executable, *Event*

Filter filters out the events that are not relevant to the slicing criterion and passes only the relevant events to *Forward Architecture Slicer*. That is, only the events that are relevant to the *slicing criterion* are fed into the *Forward Architecture Slicer*. The trace of filtered event is a finite *ordered* set of events.

(C.4) *Forward Architecture Slicer*. *Forward Architecture Slicer* computes architecture slice dynamically during the execution of the implemented architecture by following the components and ports that are visited by those filtered events and the conditions that trigger the events. In other words, a dynamic architecture slice is computed by tracing the components and ports where filtered events are generated or received according to the given slicing criterion.

(D) *Resulting Architecture Slice*. When the slicing criterion is satisfied, the slice computed up to that point is the resulting *forward dynamic software architecture slice*. It is a subset of the architecture that consists only of the components and ports that are relevant to the given slicing criterion.

We adopted the notion of *dependence table* and software architecture diagram in [6] and extended it so as to use *parameters* between components. The use of the *guards* of the events is also introduced to increase the expressive power of describing dynamic behavior of the software architecture.

3.2 The DSAS Algorithm

The algorithm computes architecture slices in forward manner, which means that it computes slices as the target program executes.

During the execution of the program, the algorithm maintains a set of components and connectors where the current event is control dependent on. That is, the set of components and connectors is necessary for the current event to happen. The event number is incremented by one whenever an event occurs.

When multiple events happens at a port, there exists a cycle of events that can be removed from the current set as the control moves to the next event. The terms and data structure used in the algorithm are as follows:

- num_sc: the event number specified in the slicing criterion
- $current\_num$: the current event number
- TES: a temporary event set along with event number and *component*
- $Left(TES)$: a function that returns the events in TES
- $Slice(comp : ev)$: an architecture slice for an event ev in a component $comp$

The formal description of the algorithm is as follows:

Input:
software architecture, $\mathcal{A}$
slicing criterion, $\mathcal{C}^d_{\mathcal{A}} = (I,\ comp : ev,\ num\_sc)$
Output:
software architecture slice, $\mathcal{S}_{\mathcal{A}}$
Algorithm:
1. **for each** event ev in the target architecture
   $Slice(comp : ev) := \emptyset$;
2. $current\_num := 0$;
3. $TES := \emptyset$;
4. **do**
   perform a statement in ADL;
   if an event ev occurs
   $current\_num := current\_num + 1$;
   $TES := TES \cup\ < comp : ev,\ current\_num >$;
   if $ev \in Left(TES)$
   let ev' be the event in $Left(TES)$ and i be its corresponding event number;
   $TES := TES \setminus \{< comp : ev',\ num > |\ i \leq num \leq current\_num\}$;
   $Slice(comp : ev) := TES$;
   while $(num\_sc \neq current\_num)$;
5. **print** $Slice(comp : ev)$;

4 Illustration

Returning to the example shown earlier in Figure 1, we show how to compute the architecture slice for *printer subsystem* example.

Figure 2 shows the relationships between the phases of the DSAS methodology and the various diagrams used in our illustration. The computation of the *forward dynamic architecture slice* of printer subsystem with respect to the *slicing criterion* can be described as follows[1]:

Software Architecture for *print subsystem* is described by a diagram as shown in Figure 1 and then the architecture is implemented using some ADL of choice In this paper, we use RAPIDE as an ADL with no specification of any internal constructs. The implemented architecture is then compiled to make an executable.

[1]We rename the components and connectors for the sake of simplicity as shown in Table 2.

$$\underbrace{\overbrace{\texttt{C1 : P1}}^{outport}\ \#\ \overbrace{\texttt{C2 : P4}}^{inport}}_{Connector}$$

where # is a delimiter, `C1:P1` is an *outport 1* of *component 1*, and `C2:P4` is an *inport 4* of *component 2*.

COMPONENT	SYMBOL	PORT	SYMBOL
User	C1	send_job	C1:P1
		prt_error	C1:P2
		start	C1:P3
		print_done	C1:P4
Queue	C2	found_error	C2:P1
		deq	C2:P2
		get_next	C2:P3
		enq	C2:P4
LPD	C3	req_job	C3:P1
		ps_file	C3:P2
		ascii_file	C3:P3
		start_job	C3:P4
		prt_ready	C3:P5
PS_Flt_D	C4	psfile_rdy	C4:P1
		psfile_in	C4:P2
PS_Prt_D	C5	ps_done	C5:P1
		start_ps	C5:P2
ASCII_Flt_D	C6	afile_rdy	C6:P1
		afile_in	C6:P2
ASCII_Prt_D	C7	ascii_done	C7:P1
		start_asc	C7:P2

Table 2. Names simplification for the components and ports of the example

EVENTS	COMPUTED ACTIVE SLICE			
	1	2	3	4
C1:P1#C2:P4	C1:P1 C2:P4			
C1:P1#C5:P2				
C1:P1#C7:P2				
C2:P1#C1:P2				
C2:P2#C3:P4		C2:P2 C3:P4		
C3:P1#C2:P3				
C3:P2#C4:P2				
C3:P3#C6:P2			C3:P3 C6:P2	
C3:P3#C7:P2				
C4:P1#C5:P2				
C5:P1#C3:P5				
C5:P1#C1:P4				
C6:P1#C7:P2				C6:P1 C7:P2
C6:P1#C4:P2				
C7:P1#C3:P5				
C7:P1#C1:P4				

Table 3. Architecture slice for the printer subsystem

When *Forward Dynamic Architecture Slicer* starts it gets *printer subsystem* architecture and slicing criterion as input and builds table similar to Table 2 and the ADL executable generates a *bag of events*.

In slicing criterion, we assume that *printer subsystem* architecture has five users and we are interested in the fourth event that occurred at the *inport* of *PS_Flt_D* component and the *user id* 2 The *initial values* for *job* and *user id* are set to be ascii_ps.txt and 2 respectively. The *event name* is C4 : P2 and its event number is 4. Then the slicing criterion becomes

$$\mathcal{C}^d_{\mathcal{A}} = ((\text{ascii\_ps.txt}, 2),\ \text{C4} : \text{P2},\ 4).$$

In other words, we are interested in the *fourth event* (4) at the *inport* psfile_in (P2) of component PS_Flt_D (C4) and initial values for *job* is "ascii_ps.txt", and *user id* is 2.

Event Filter receives the events from the ADL executable. It filters out the events that are irrelevant to the slicing criterion and produces a *set of events* that is relevant to the *Slicing Criterion*.

Forward Architecture Slicer gets the *Software Architecture* as input and the filtered events from *Event Filter*. For each occurrence of a filtered event, *Forward Architecture Slicer* computes a new architecture slice.

In Table 3, the first column lists the *connectors* in the architecture and each cell holds partial slice (*ports*) that are used by an event whose event number is the corresponding column number. The architecture slice is recomputed whenever a filtered event is received. The architecture slice at a given time is the union of the ports in the Table 3.

For example, when *Forward Architecture Slicer* receives a filtered event with event number 2, the architecture slice at that point is:

$$\begin{aligned} &\{< \text{C1} : \text{P1},\ 1 >, < \text{C2} : \text{P4},\ 1 >\} \bigcup \\ &\qquad \{< \text{C2} : \text{P2},\ 2 >, < \text{C3} : \text{P4},\ 2 >\} \\ &= \quad \{< \text{C1} : \text{P1},\ 1 >, < \text{C2} : \text{P4},\ 1 >, \\ &\qquad < \text{C2} : \text{P2},\ 2 >, < \text{C3} : \text{P4},\ 2 >\}. \end{aligned}$$

When the given *slicing criterion* is satisfied, *Forward Dynamic Slicer* stops and prints the resulting slice. The resulting software architecture slice contains only the *components* and *ports* that are relevant to the slicing criterion. The resulting slice, according to Table 3, is as follows:

$$\begin{aligned} \mathcal{S}_{\mathcal{A}} = \{&\text{C1} : \text{P1},\ \text{C2} : \text{P4},\ \text{C2} : \text{P2},\ \text{C3} : \text{P4}, \\ &\text{C3} : \text{P3},\ \text{C6} : \text{P2},\ \text{C6} : \text{P1},\ \text{C7} : \text{P2}\}. \end{aligned}$$

The resulting printer subsystem architecture slice is shown in Figure 3. The grayed-out portion of the diagram is not the part of the architecture slice.

5 Conclusion

In this paper, we have introduced the notion of *dynamic software architecture slicing* (*DSAS*) to facilitate the understanding and analysis of software architecture. This notion hinges on only a couple of intuitive concepts, namely, dynamic software architectural slice and dynamic software ar-

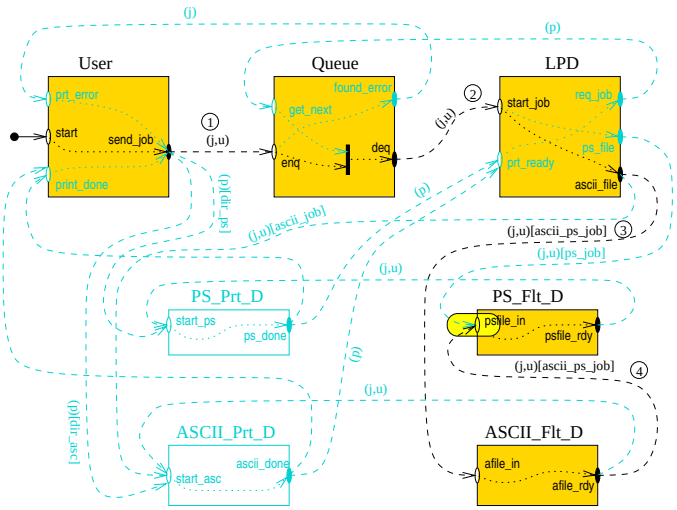

Figure 3. Dynamic architecture slice of printer subsystem

chitecture slicing criterion, and yet it goes beyond the notion of static software architecture slicing while exploiting the benefits of the "dynamic" aspect of dynamic program slicing.

Given a software architecture, a dynamic software architecture slice represents the particular sequence of components and connectors of the architecture, to be involved in the execution of a program which implements it, with respect to a particular set of events and variable-value bindings, i.e. a particular software architectural slicing criteria. Compared to static software architecture slicing, dynamic slicing generates potentially a much smaller number of components and connectors in each slice especially when there are a large number of ports whose invocation depends on change in the value of a variable or the triggering of an event.

In this paper, we have also presented a methodology to use the notion and an algorithm to generate dynamic architecture slice.

Being the first in its kind, however, our proposal needs improvements along more than one avenue. One such avenue concerns the completeness of our formalization of the notion of dynamic software architecture slicing. Work is underway to augment the current set of definitions with, for example, the precise relationship between a dynamic architectural slice and its corresponding program slice. Another avenue of future work concerns the consideration of several different types of slicing criterion in relation to, for example, fault detection and tolerance and (perhaps weaker notions of) deadlock and livelock.

Notwithstanding these issues, we feel that we now have a basis for developing other more powerful varieties of dynamic software architecture slicing as a promising technique for a wide variety of software engineering activities, such as detection of specification errors, reverse engineering and re-engineering, reuse and fault localization.

References

[1] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[2] M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, July 1984.

[3] H. Agrawal and J. R. Horgan. Dynamic program slicing. *Proc. ACM SIGPLAN'90*, pages 246–256, 1990.

[4] B. Korel and S. Yalamanchili. Forward computation of dynamic program slices. *ISSTA 94, Seattle Washington*, pages 66–79, 1994.

[5] Y. Song and D. T. Huynh. Forward dynamic object-oriented program slicing. *Proc. Application Specific Systems and Software Engineering '99*, March, March 1999.

[6] 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.

[7] J. Zhao. Slicing software architectures. *Technical Report 97-SE-117, Information Processing Society of Japan*, pages 85–92, November 1997.

[8] D. Garlan, R. Monroe, and D. Wile. Acme: An architecture description interchange language. *Proc. CASCON '97*, 1997.

[9] D. C. Luckham, J. J. Kenney, and L. M. Augustin. Specification and analysis of system architecture using rapide. *IEEE Trans. Software Engineering*, 21(4):336–345, April 1995.

[10] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. *Proc. ACM SIGSOFT*, pages 179–185, December 1994.

[11] 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*, 21(4):314–335, April 1995.

[12] R. Allen and D. Garlan. Formalizing architectural connection. *Proc. 16th International Conference on Software Engineering*, pages 71–80, May 1994.