Uncertainty Problem in Dynamic Slicing of Concurrent Programs

Jingde Cheng
Department of Information and Computer Sciences
Saitama University, Saitama, 338-8570, Japan
cheng@ics.saitama-u.ac.jp

“As long as a branch of science offers an abundance of problems, so long is it alive: a lack of problems foreshadows extinction or the cessation of independent development.”
-- David Hilbert, 1900.

Abstract

Although both static and dynamic slicing of sequential programs have been applied to software engineering practices, there still are some challenges, problems, and issues in slicing concurrent programs, in particular, dynamic slicing of concurrent programs. This paper presents a methodological review of dynamic slicing methods for concurrent programs, points out that the most intrinsic problem in all existing dynamic slicing methods is the uncertainty problem, and shows a new research direction for dynamic slicing of concurrent programs. The paper proposes two basic criteria, i.e., completeness and soundness, for dynamic slicing of concurrent programs, and shows that we should develop a dependence/influence analysis method based on self-measurement principle to obtain complete and sound dynamic slices of concurrent programs.

1. Introduction

Program dependences are dependence relationships holding between statements in a program that are implicitly determined by control flows and data flows in the program. Program dependence analysis is a program analysis technique to identify and determine various program dependences implied in program source codes and then represent them in some explicit forms convenient for various applications [3, 6-10, 31, 42].

Program slicing, originally introduced by Weiser [47-49], is a program decomposition technique that reduces a target program, by analyzing program dependences between statements in the program, into a minimal form, called a slice of the original target program, such that the slice preserves, according to some given criterion, some behavior of the original program. Tip gave the first survey on this subject [45] and there are some other surveys [17, 22, 39, 50].

Today, program slicing has been developed as a standard technique used in various software engineering activities including program understanding, testing, debugging, maintenance, and complexity measurement. However, although both static and dynamic slicing of sequential programs have been applied to software engineering practices, there still are some challenges, problems, and issues in slicing concurrent programs, in particular, dynamic slicing of concurrent programs. Therefore, we have a fundamental question: Where is the right way for dynamic slicing of concurrent programs?

The question, “Where is the right way?” invites the immediate counter-question “Right for what?” Only if we certainly know what we want to obtain, we can make a good choice. This paper presents a methodological review of dynamic slicing methods for concurrent programs from the viewpoint of wholeness, uncertainty, and self-measurement principles of concurrent systems, points out that the most intrinsic problem in all existing dynamic slicing methods is the uncertainty problem, and shows a new research direction for dynamic slicing of concurrent programs.

The rest of the paper is organized as follows: Section 2 presents some intrinsic characteristics of concurrent systems and/or programs, Section 3 presents a methodological review of dynamic slicing methods for concurrent programs, and points out the uncertainty problem in all existing dynamic slicing methods, Section 4 proposes two basic criteria, i.e., completeness and soundness, for dynamic slicing of concurrent programs from the viewpoint of applications in software engineering, and shows that we should develop a dependence/influence analysis method based on self-measurement principle to obtain complete and sound dynamic slices of concurrent programs.
programs, and some concluding remarks are given in Section 5.

2. Intrinsic Characteristics of Concurrent Systems and/or Programs

Concurrent systems, i.e., parallel or distributed computing systems, are becoming more and more widely used [2, 5]. Therefore, modern society is more and more dependent on concurrent systems, and, of course, dependent on the reliable and secure functioning of the systems. On the other hand, a good design, development, and maintenance methodology for concurrent systems must be based on a deep recognition and understanding of the intrinsic characteristics of concurrent systems.

According to IEEE Standard 610 Computer Dictionary, the term ‘concurrent’ means “Pertaining to the occurrence of two or more activities within the same interval of time, achieved either by interleaving the activities or by simultaneous execution.” According to the same standard, the term ‘system’ means “A collection of components organized to accomplish a specific function or set of functions.”

In general, from the viewpoint of structure, a concurrent system consists of a number of processes with some organized structure to accomplish a specific function. Each of the processes is a sequence of activities occurring in such a manner that one must finish before the next begins, and therefore, in its lifetime, a process forms a thread of control flow and/or a thread of data flow. The multiple threads of control flow and multiple threads of data flow in a concurrent system are not independent because of the existence of interprocess synchronization among multiple threads of control flow and interprocess communication among multiple threads of data flow in the system. Moreover, a process in a concurrent system may nondeterministically select a communication partner among a number of processes ready for communication with that process. Any activity and/or behavior of a process must directly or indirectly, deterministically or nondeterministically influence other processes through the interprocess synchronization and communication. Therefore, it is the interprocess synchronization and communication in a concurrent system that form the organized structure of the concurrent system.

From the viewpoint of information exchange, almost concurrent systems, e.g., operating systems, real-time process control systems, or database management and transaction processing systems, are open systems, i.e., they run in a way such that they have intensive interactions with their environments and therefore they can selectively exchange information with their environments. Sometimes it is convenient and even necessary to regard a user and/or client of an open system as a part of the system.

From the viewpoint of behavior, in general, no process of a concurrent system can be created, changed, modified, or removed without the impact of behavior change on other processes of the system. A variation of any process of a concurrent system must directly or indirectly, deterministically or nondeterministically affect all other processes of the system and brings about a variation in the whole system, and vice-versa, variations of any process also depend upon all other processes of the system. For a concurrent system, in particular, an open concurrent system, we cannot adequately observe properties of the whole system from an analysis of just the parts apart. For example, a deadlock or livelock, which is generally regarded as an undesirable faulty state, among processes in a concurrent system is such a phenomenon that cannot be observed or analyzed in every process one by one but only can observed or analyzed from the interprocess interactions in the system.

An intrinsic characteristic of concurrent programs is the so-called “unreproducibility of behavior,” i.e., for a concurrent program, two different executions with the same input may produce different behavior and histories because of unpredictable rates of processes and existence of nondeterministic selection statements in the program [21, 33, 34, 43].

Therefore, unlike sequential programs where any program behavior is value-dependent, the behavior of concurrent programs is usually not only value-dependent but also timing-dependent. It is obvious that when we investigate a concurrent program, if we focused our attentions only on its value-dependent behavior, then we have lost some intrinsic characteristics of that program.

An important principle in systems science is the wholeness principle: “the whole is more than the sum of parts” is simply that constitutive characteristics are not explainable from the characteristics of isolated parts [46].

Based on the above discussion, from the viewpoint of systems science, we can say that the most intrinsic behavioral characteristic of a concurrent system is its wholeness. The present author has proposed the following principle [11, 12]:

The wholeness principle of concurrent systems: “The behavior of a concurrent system is not simply the mechanical putting together of its parts that act concurrently but a whole such that one cannot find some way to resolve it into parts mechanically and then simply compose the sum of behavior of its parts as the same as its original whole behavior.”
Based on the above proposition, we can say that the most intrinsic difference between a sequential system and a concurrent system is the wholeness of behavior, i.e., the whole behavior of a sequential system is simply integratable while the whole behavior of a concurrent system is not. The whole behavior of a sequential system is just the sum of behavior of its components and can be observed or analyzed from its components apart and sequentially. While the whole behavior of a concurrent system can only be observed or analyzed from the organized structure of its all components. A sequential system can be implemented by either a group of sequential programs or a group of parallel or distributed programs. While a concurrent system can only be implemented by a group of parallel or distributed programs.

**Measuring** the behavior of a computing system means capturing run-time information about the system through detecting attributes of some specified objects in the system in some way and then assigning numerical or symbolic values to the attributes in such a way as to describe the attributes according to clearly defined rules. **Monitoring** the behavior of a computing system means collecting and reporting run-time information about the system, which are captured by measuring the system. Measuring and monitoring mechanisms can be implemented in hardware technique, or software technique, or both.

A **target system** is a computing system that is the target in measuring and monitoring. The **target programs** are a set of program source codes that implemented the target system. The **target programming languages** is a set of the programming languages that are used to describe the target programs. A **measurement point** is a location in a target program where an event concerning some attribute of an object, which is specified by measuring requirements, occurs during an execution of the program. A measurement point generally has a semantic interpretation at the target programming language level. A **sensor** is a detector, which may be implemented in either hardware technique or software technique, such that it detects the event occurred at a measurement point in the target programs when they are executed, and assign a numerical or symbolic value to the attribute that corresponds to the event. In general, a sensor implemented in software technique is a section of code, which is inserted within a measurement point of a target program. In this case, the program obtained from a target program by inserting sensors is called a **measured and monitored program**.

Almost all existing computing systems are designed, developed, and maintained in a **separate-measurement** methodology, i.e., a computing system is constructed at first only by some function components and there is no self-measurement component as permanent non-functional components [2, 4, 19]. As a result, when we want to measure and monitor the behavior of the target system, we have to use a separate measuring and monitoring (hardware or software) tool that inserts sensors into the target system and collects run-time information of the system when it is executed for measuring and monitoring. In general, the separate measuring and monitoring tool is independent on the target system.

In measuring and monitoring the behavior of a computing system by the separate-measurement approach, the detection activities of sensors, no matter how they are implemented in whatever mechanism, will interfere with program performance of the target system.

The separate-measurement approach is effective and efficient to the development and maintenance of sequential systems. This because that the whole behavior of a sequential system is just the sum of behavior of its components and can be observed or analyzed from its components apart and sequentially, and therefore, the behavior of an observer such as a run-time monitor can be simply separated from what is being observed, even though the detection activities of sensors interfere with program performance of the target system.

However, measuring and monitoring the behavior of a concurrent system by the separate-measurement approach is not effective to the development and maintenance of concurrent systems. In measuring and monitoring the behavior of a concurrent system by the separate-measurement approach, the detection activities of sensors, no matter how they are implemented in whatever mechanism, will interfere with program performance of the target system and must ultimately modify the whole behavior of target programs. This is the “Heisenberg uncertainty principle” as applied to measuring and monitoring [21, 33, 34].

As a direct result of the wholeness principle of concurrent systems, we can recognize and understand the following principle [11, 12]:

**The uncertainty principle in measuring and monitoring concurrent systems:** “In measuring and monitoring a concurrent system, the behavior of a observer (a measurer or a monitor) cannot be separate from what is being observed.”

When we use the separate-measurement approach in measuring and monitoring the behavior of a concurrent system, what a run-time monitor can report for us is the run-time information about the system that is being measured and monitored, i.e., the system with influences from sensors, but not the target system. The target system without influences from sensors may act
in some behavior that is very different from that the run-time monitor reported. For example, some deadlocks or livelocks may occur in a concurrent system without influences from measuring and monitoring, even though the deadlocks and livelocks have not occurred in measuring and monitoring the system.

According to the wholeness principle of concurrent systems and the uncertainty principle in measuring and monitoring concurrent systems, we now know that it is impossible to completely eliminate the interference with the behavior of the measured and monitored systems caused by measuring and monitoring. Therefore, the most important challenging problem in measuring and monitoring concurrent systems is how to deal with the interference.

In measuring and monitoring concurrent systems, besides the separate-measurement approach, a self-measurement approach is also possible. In the self-measurement approach, the target system is designed and developed in a way such that all sensors, which also may be implemented in either hardware technique or software technique, for detecting events in target system are previously prepared and the sensors remain in the system permanently. In this approach, the measuring and monitoring activities are always a part of the whole behavior of the target system itself.

Almost all existing concurrent systems are designed, developed, and maintained following the separate-measurement methodology. However, as we have pointed out, it is even impossible to grasp the “true” and “pure” behavior of target systems in this way. Based on the wholeness principle of concurrent systems and the uncertainty principle in measuring and monitoring concurrent systems, the present author has proposed a new design principle for concurrent systems as follows [11, 12]:

The self-measurement principle in designing, developing, and maintaining concurrent systems: “A large-scale, long-lived, and highly reliable concurrent system should be constructed by some function components and some (maybe only one) permanent self-measurement components that act concurrently with the function components, measure and monitor the system itself according to some requirements, and pass run-time information about the system’s behavior to the outside world of the system.”

3. Dynamic Slicing of Concurrent Programs: A Methodological Review

A static slice of a concurrent program on a given static slicing criterion (typically, a statement s in the program and a set V of variables used at s) consists of all parts in the program that possibly affect the beginning or end of execution of a program point of interest (e.g., statement s), or possibly affect the values of variables (e.g., variable set V) at the program point [8]. A static slice is computed by static dependence analysis without making assumptions regarding a program’s input and therefore is valid for all possible executions of the program.

Dynamic slicing of programs was originally introduced by Korel and Laski [27, 28]. In the case of dynamic slicing of programs, unlike the case of static slicing of programs where all dependences that occur in all possible executions of the program are considered, only the dependences that occur in a specific execution of the program are taken into account.

There are some different definitions for the notion of dynamic slice proposed based on different motivations [1, 6-10, 18, 20, 23-30, 32, 35-39, 44].

Originally, Korel and Laski defined a dynamic slice (of a sequential program) as “an executable part of the program whose behavior is identical to that of the original program with respect to a subset of variables of interest and at execution position q.” [25-30]

Agrawal and Horgan defined a dynamic slice (of a sequential program) as follows: “Given an execution history hist of a program P for a test-case test, and a variable var, the dynamic slice of P with respect to hist and var is the set of all statements in hist whose execution had some effect on the value of var as observed at the end of the execution.” [1]

The present author first introduced three types of primary program dependences in concurrent programs, named the “selection dependence,” “synchronization dependence” and “communication dependence,” and a program representation, named “Process Dependence Net” for concurrent programs [6-10]. The present author discussed various possible applications of Process Dependence Net including slicing concurrent programs that first extended the notion of slicing sequential programs into the case of concurrent programs [6-10]. The present author defined a dynamic slicing criterion and a dynamic slice of a concurrent program as follows: “A dynamic slicing criterion of a concurrent program is a quadruplet (s, V, H, I), where s is a statement in the program, V is a set of variables used at s, and H is a history of an execution of the program with input I. The dynamic slice DS(s, V, H, I) of a concurrent program on a given dynamic slicing criterion (s, V, H, I) consists of all statements in the program that actually affected the beginning or end of execution of s and/or affected the values of variables in V at s in the execution with I that produced H.”

244
Duesterwald, Gupta, and Soffa introduced an execution trace representation, called the “Distributed Dependence Graph” for distributed programs, and showed a parallel algorithm to compute dynamic slices of a distributed program based on its distributed dependence graph representation [18]. Duesterwald, Gupta, and Soffa defined a distributed dynamic slicing criterion and a dynamic slice of a distributed program as follows: “Given a distributed program \( P = (P_1, P_2, ..., P_n) \), a distributed dynamic slicing criterion for \( P \) is a tuple \( C = (I_1, X_1, I_2, X_2, ..., I_n, X_n) \), where \( I_i \) is the input to process \( P_i \) and \( X_i \) is a set of statements in \( P_i \). Given a slicing criterion \( C = (I_1, X_1, ..., I_n, X_n) \) for a distributed program \( P = (P_1, P_2, ..., P_n) \), a distributed dynamic slice \( S = (P'_1, P'_2, ..., P'_n) \) is an executable subset of \( P \), such that \( P'_i \) is a subset of \( P_i \) and when \( S \) is executed on input \( (I_1, ..., I_n) \) it produces the same values for the variables in \( (X_1, ..., X_n) \) as \( P \) does.”

Korel and Ferguson proposed a method to compute dynamic slices of a distributed Ada program by analyzing influences in and between multiple executed paths of the program [26]. Korel and Ferguson defined that “A dynamic slicing criterion of distributed (Ada) program \( Q \) is a quintuple \( C = (x, P, w, q, v) \), where \( P \) is a distributed program path which has been traversed during execution of program \( Q \) on input \( z, q \) is a position in path \( L_w \) of task \( w \), and \( v \) is a variable in task \( w \).” They defined a dynamic slice of a distributed Ada program as “an executable part of the program whose behavior is identical to that of the original program with respect to a variable (or a set of variables) of interest and at execution position \( q \) in task \( w \).” See [26] for their formal definition.

Following the works mentioned above, the problem of dynamic slicing of concurrent programs has drawn the attention of many researchers. However, those following works only extended or modified program representations and dynamic slicing algorithms used in the early works but did not make any novel methodological development.

Kamkar and Krajina introduced, as a modification of “Process Dependence Net” and “Distributed Dependence Graph”, a program representation, called “Distributed Dynamic Dependence Graph” that represents control, data, and communication dependences in a distributed program. The graph is built at run-time and it is used to compute dynamic slices of the program [23, 24]. Kamkar and Krajina defined a dynamic slice of a distributed program as follows: “Given a distributed program \( P = \{p_1, p_2, ..., p_n\} \), where \( p_i \) is a process in \( P \), we define a dynamic slice on program \( P \) to be \( \text{ProgSlice} = \{s_1, s_2, ..., s_m\} \) where \( s_i \) is a subset of statements of process \( p_i \) and contains all statements which directly or indirectly, under a particular execution of the program, affect the computation of specified statement(s) in the slicing criterion. A distributed slicing criterion is a non-empty set defined to be : \( C = \{s_i, x_i\} \) where \( s_i \) is a selected statement in process \( P_i \) with test case \( x_i \).”

Goswami and Mall introduced, as a modification of Process Dependence Net and Distributed Dependence Graph, the notion of a “Dynamic Program Dependence Graph” to represent various intraprocess and interprocess dependences of concurrent programs [20].

Mohapatra, Mall, and Kumar have introduced the notion of “Distributed Program Dependence Graph” as the program representation for dynamic slicing of concurrent object-oriented programs [35-38].

Tallam, Tian, and Gupta have extended Dynamic Program Dependence Graph to include selected Write-After-Write and Write-After-Read dependences and developed an extended dynamic slicing algorithm working based on the extended Dynamic Dependence Graph to detect data races in concurrent programs [44].

We can see that a major difference among the above definitions of dynamic slice is whether or not to require a dynamic slice to be executable. Here, the key point is the purpose to slice a program, i.e., why one want to obtain dynamic slices of a program, or for what the slice will be used. If “executable” is required for executing the slice time after time, then what is the purpose of “executing the slice time after time”? If “executable” and “executing the slice time after time” only are a way to achieve some purpose, then once we have some way to achieve the same purpose, we do not need to require that a dynamic slice must be executable. For example, from the viewpoint to regard debugging a program as the process of locating, analyzing, and ultimately correcting bugs in the program by reasoning about causal relationships between bugs and the errors which have been detected in the program, if we have sufficient information about a program, its specification, and an execution of it where an error is detected, then, in principle, we should be able to debug the program by analyzing the error and reasoning about bug(s) caused the error based on our information, knowledge, and experiences without totally or partially executing the program time after time. Therefore, from the viewpoint of debugging, we should focus our attention on whether or not a slice of a program is adequate to a debugging problem rather than whether or not a slice of a program is executable.

Another major difference among the above definitions concerning dynamic slices of concurrent programs is whether or not to take the timing-dependent behavior of a concurrent program into account. The present author pointed out the importance of considering timing-dependent behavior in slicing concurrent programs and explicitly described
timing-dependent behavior issue in his definition of slices of concurrent programs. It is obvious that Duesterwald, Gupta, and Soffa only considered value-dependent behavior of a distributed program but did not take the timing-dependent behavior of the program into account. Korel and Ferguson discussed timing-dependent behavior issue in their definition. In the vague definition by Kamkar and Krajina, what “affect the computation of specified statement(s)” means is not explicit.

Note that without regard to define a dynamic slice of a concurrent program to be executable or not, all the above definitions for dynamic slicing require an execution history of “the target program” as a parameter. In fact, it is this execution history that is most crucial to the problem of dynamic slicing of concurrent programs.

The three fundamental principles of concurrent systems and/or programs we presented in Section 2 imply the following propositions.

First, according to the wholeness principle, for any concurrent program, there is no set of executable dynamic slices such that the sum of their behavior is the same as the original whole behavior of the program.

Second, according to the uncertainty principle, since in measuring and monitoring a concurrent program, the behavior of a measurer and/or monitor cannot be separate from what is being observed, any execution history reported by the measurer and/or monitor must be the behavior of the measured and monitored program but not the original behavior of the target program. According to both the wholeness principle and the uncertainty principle, it is impossible to grasp the “true” and “pure” behavior of the target program in the separate-measurement way of measuring and monitoring. Therefore, if one use the separate-measurement way to get execution histories of a concurrent program for dynamic slicing, then the obtained dynamic slices must be that of the measured and monitored program but not ones of the original target program.

Finally, according to the self-measurement principle, if we do not at first expect to obtain the “true” and “pure” execution history of the target program but want to obtain sufficient information for, say, debugging, then using the self-measurement way of measuring and monitoring should be able to provide us the information.

All existing dynamic slicing methods for concurrent programs are the same in that any one is based on a separate-measurement methodology, i.e., to measure and monitor at first the behavior of a concurrent program in a specific execution using a separate measuring and monitoring tool that is independent on the target program, and then to compute dynamic slice of the target program using the execution history recorded in monitoring. However, according to the three fundamental principles of concurrent systems and/or programs, it is even impossible to grasp the “true” and “pure” behavior of the target program in this way. Therefore, no one of the existing dynamic slicing methods for concurrent programs can compute a “true” and “pure” dynamic slice of target programs. All discussions on the subject in fact only focus attentions on the behavior of the programs being monitoring that just is a “counterfeit” of the original behavior of target programs.

Consequently, this uncertainty problem in dynamic slicing of concurrent programs is open until now.

4. Dynamic Slicing of Concurrent Programs: Where is the Right Way?

The effectiveness or usefulness of dynamic slices of a sequential program is intrinsically an implication of the behavior reproducibility of the program. While, since concurrent programs do not have the behavior reproducibility, it is meaningless and useless to apply the results of analyzing and understanding dynamic slices obtained in a specific execution of a target program to other different executions of the program.

Program slicing should be a technical way for achieving some purposes in software engineering but not purposes themselves. From the viewpoint of applications in software engineering, we propose the following two basic criteria, Completeness and Soundness, for dynamic slicing of concurrent programs:

Completeness: For a class of concurrent programs and a certain purpose of application, if a dynamic slice of a program in the class computed by a dynamic slicing method can provide sufficient information for the purpose of application, then the dynamic slice is said to be complete for the purpose of application; a dynamic slicing method is said to be complete for a class of concurrent programs and a certain purpose of application if and only if any dynamic slice of any program in the class computed by the dynamic slicing method is complete for the purpose of application. (Note that the trivial case, the program itself, is obviously complete for any purpose of any application, but not meaningful and useful in the sense of program slicing.)

Soundness: For a class of concurrent programs and a certain purpose of application, if a dynamic slice of a program in the class computed by a dynamic slicing method provide no counterfeit information for the purpose of application, then the dynamic slice is said to be sound for the purpose of application; a dynamic slicing method is said to be sound for a class of
concurrent programs and a certain purpose of application if and only if any dynamic slice of any program in the class computed by the dynamic slicing method is sound for the purpose of application.

Only if a dynamic slicing method for concurrent programs is both complete and sound for any class of concurrent programs and any reasonable purpose of application, we can say that dynamic slicing method is the really right method for dynamic slicing of concurrent programs. Unfortunately, no one of all existing dynamic slicing methods for concurrent programs is both complete and sound for any class of concurrent programs and any reasonable purpose of application.

Based on the three fundamental principles of concurrent systems and/or programs, if our original purpose to dynamically slice a concurrent program is to obtain sufficient information about those program fragments that affect behavior of the program at a program point of interesting in a specific execution of the program rather than a reduced executable subprogram, then we should develop a dependence/influence analysis method based on the self-measurement principle such that to collect useful information at run-time during the normal execution of the program by the self-measurement way of measuring and monitoring, and then to filter those fragments from the program by dependence/influence analysis. We can still call these program fragments “dynamic slices” of the target program, if we like, even if they are not necessarily executable. This method may be the only hopeful way to obtain complete and sound dynamic slices of concurrent programs. Refer [40, 41] to see what primary works have been done based on the self-measurement principle.

5. Concluding Remarks

Recently, concurrent systems are more and more large and complex. Almost all practical concurrent systems are large-scale distributed systems. The timing-dependent behavior and its un reproduciability is the rule rather than the exception for any large-scale distributed system. Therefore, how to recognize, consider, investigate, and ultimately solve the problem of dynamic slicing of concurrent programs is a big challenging engineering issue.

On the other hand, as a novel computing paradigm, “Persistent Computing” aims to design and develop persistent computing systems which are continuously dependable and dynamically adaptive in the sense that (1) the systems can function continuously and persistently without stopping its reactions to its outside environment, and (2) the systems can be dynamically maintained, upgraded, or reconfigured during its continuous functioning and reacting [13-16]. For a persistent computing system, the problem of dynamic slicing may disappear if all run-time information needed in any engineering activity can be completely and soundly provided by the persistent computing system itself at anytime.

References