University of Illinois at Urbana-Champaign
Department of Computer Science Technical Report UIUCDCS-R-2005-2660

Predicting Concurrency Errors at Runtime using Sliced Causality

Feng Chen and Grigore Roşu

Department of Computer Science
University of Illinois at Urbana - Champaign, USA
{fengchen,grosu}@uiuc.edu

Abstract. A runtime analysis technique is presented, which can predict errors in multi-threaded systems by examining event traces generated by executions of these systems even when they are successful. The technique is based on a novel partial order relation on relevant events, called *sliced causality*, which loosens the obvious but strict "happens-before" relation by considering static structural information about the multi-threaded program, such as control-flow and data-flow dependence, and dynamic synchronization information, such as lock-sets. A vector clock based algorithm to encode the sliced causality is given, together with a procedure for generating all potential runs that are consistent with this partial order in a memory effective way. Then violations of properties can be "predicted" by running the corresponding monitor against *potential runs* that are consistent with the observed execution, i.e., permutations of (abstract) events that do not violate the sliced causal partial order. The monitors can be manually implemented or automatically synthesized from the desired properties, which can be given in any formalism that allows monitor synthesis algorithms. Our runtime analysis technique is *sound*, in the sense that it reports no false alarms. As expected, it is *not* complete; indeed, it cannot say anything about code that was not reached during the observed execution. A prototype system, called JPREDICTOR, has been implemented and evaluated on several Java applications with promising results.

1 Introduction

A characteristic of concurrent systems in general and of multi-threaded systems in particular is that the same program with the same input may exhibit different behaviours when executed at different times. This inherent nondeterminism makes multi-threaded programs difficult to analyze, test and debug. This paper introduces a technique to correctly detect concurrency errors from observing execution traces of mutlithreaded programs. The program is automatically instrumented to emit "more than the obvious" information to an external observer, by means of runtime events. The particular execution that is observed needs *not* hit the error; yet, errors in other executions can be predicted *without false alarms*. The observer, which can potentially run on a different machine, will never need to see the code which generated those events but still be able to correctly predict errors that may appear in other executions, and report them to the user in a meaningful manner, by counter-example executions that explicitly reveal those errors.

There are several other approaches in the literature also aiming at detecting potential errors in concurrent systems by examining particular execution traces. Some of these approaches aim at verifying general purpose behavioral properties [27, 28], including temporal ones, and are inspired from efforts in debugging distributed systems based on Lamport's "happens-before" causal partial ordering on runtime events [18]. Other approaches aim at dynamic behavior reduction and have been designed to work best for particular properties of interest, such as data-race and/or atomicity detection by means of lock-set algorithms [25, 14]. These previous efforts focus on either soundness or coverage: approaches based on the "happen-before" relation are sound but have limited coverage over interleavings, thus resulting in more false negatives (missing errors); lock-set based approaches produce fewer false negatives but suffer from false positives (false alarms). There are also works combining "happen-before" and lock-set techniques, e.g., [21], aiming at achieving a better balance between soundness and coverage, but these focus on particular properties to check, such as data-races, and do not use static information to increase coverage.

The approach presented in this paper focuses on improving the coverage without breaking the soundness. This is achieved by combining dynamic analysis, based on a special "happen-before" causal partial order, with static control-flow and dynamic data-flow dependency information of the multi-threaded program. This results in a much loosened causal partial order relation on events, which we call *sliced causality*, because, based on an apriori step of static analysis of the program's code, it drastically cuts the usual "happen-before" causality by removing unnecessary dependencies; this way, a large number of consistent runs can be inferred and analyzed by the observer of the multithreaded execution. This novel causality relation leads to a practical technique for sound violation prediction of general-purpose properties, with significantly less coverage compromise than the other "happen-before" approaches.

One should not confuse the notion of sliced causality introduced in this paper with the existing notion of *computation slicing* [26]. The two slicing techniques are quite opposed in scope: the objective of computation slicing is to safely *reduce* the size of the computation lattice extracted from a run of a distributed system, in order to reduce the complexity of debugging, while our goal is to *increase* the size of the computation lattice extracted from a run, in order to strengthen the predictive power of our analysis by covering more consistent runs. Computation slicing and sliced causality do not exclude each other. Sliced causality can be used as a front end to increase the coverage of the analysis, while computation slicing can then remove redundant consistent runs from the computation lattice, thus reducing the complexity of analysis. At this moment we do not use computation slicing in our implementation, but it will be addressed in our future works to improve the performance of JPREDICTORin addition to the use of more efficient instrumentation algorithms.

Our predictive runtime analysis technique can be understood as a hybrid of testing and model checking. Testing because one runs the system and observes its runtime behavior in order to detect errors, and model checking because the special causal partial order extracted from the running program can be regarded as an abstract model of the program, which can further be investigated exhaustively by the observer. To avoid false alarms, the permutations of abstract events analyzed by the observer should be consistent with the semantics of the original program. Previous approaches based on the "happen-before" idea (such as [21, 27, 28]) extract causal partial orders from analyzing *exclusively* the dynamic thread communication in program executions. Since these approaches consider *all* interactions among threads, e.g., all reads/writes of shared variables, the obtained causal partial orders are rather *restrictive*, or *rigid*, in the sense of allowing a reduced number of linearizations and thus of errors that can be detected. Note that, in general, the larger the causal order (as a binary relation) the fewer linearizations it has, i.e., the more restrictive it is. By considering information about the static structure of the multi-threaded program in the computation of the causal partial order, we can filter out irrelevant thread interactions and thus obtain a more *relaxed* causality, allowing more consistent runs. Furthermore, we also take synchronization into account in our approach, in the sense that events protected by locks can only be permuted *in blocks*. This way, our approach borrows comprehensiveness from lock-set approaches without giving-up soundness. Moreover, our approach is fully generic: the possible linearizations that are consistent with the observed causal partial orders can be checked against *any* property on execution traces.

Figure 1 shows the classical example of the limitation of the happen-before techinque [25]. When the execution proceeds as indicated by the arrow, the datarace on y is masked by the protected accesses to x because the accesses to y are ordered by the write/read of x in the traditional happen-before approach. However, one can see that the accesses to y in the thread t_2 do *not* depend on the accesses to x in the same thread; in other words, no matter what the value of x is , the accesses to y will occur anyway. Therefore, our approach drops the causal partial order caused by read/write of x and predicts the datarace on y in this successful execution.

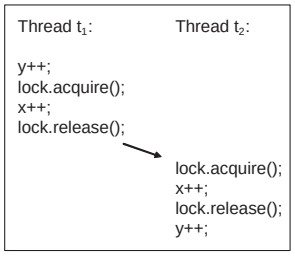

Fig. 1. Limitation of happen-before technique

Let us further explain our predictive runtime analysis technique on a more abstract example. Assume the threads and events in Figure 7, where e_1 causally precedes, or "happens-before", e_2 (e.g., e_1 writes a shared variable and e_2 reads it right afterwards), and the statement generating e'_3 is in the *control scope* (this notion will be formally defined in Section 4)[1] of the statement generating e_2, while the statement generating e_3 is not in the control scope of e_2. Then we say that e'_3 *is dependent upon* e_1, but that e_3 is *not* dependent upon e_1, despite the fact that e_1 obviously happened before e_3. The intuition here is that e_3 would happen anyway, with or without e_1 happening. Because of its combined static/dynamic flavor, we call this new depedence relation on events the *hybrid dependence*. Interestingly, if the events in the scope of e_2 are not relevant for the property to check, then any permutation/linearization of relevant events consistent with the intra-thread total order and the hybrid dependence corresponds to some valid execution of the multithreaded system. Therefore, if any of these permutations violate the property, then the system can do so in a

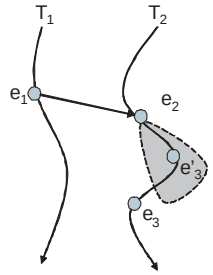

Fig. 2. Hybrid causality.

[1] For now, one can think of it as control-/data-flow dependence, e.g., e'_3 may be generated by a statement in the "then" branch of a conditional statement that previously generated the read event e_2 while evaluating its condition.

```
class MyThread extends Thread{
        static public MyLock lock;
        static public int threadCount;
        public static void main(String[] args) {
                lock = new MyLock();
                threadCount = 0;
                (new MyThread()).start();
                (new MyThread()).start();
        }
        public void run(){
                if (lock == null)
                        return;
                lock.tryAcquiring();
                MyThread.threadCount ++;
                lock.release();
        }

}
```

```
class MyLock {
        public boolean flag;
        public int count;
        public MyLock(){
                flag = false;
                count = 0;
        }
        public boolean tryAcquiring(){
                synchronized (this){
                        if (flag)
                                return false;
                        flag = true;
                }
                count ++;
                return true;
        }
        public void release(){
                synchronized (this){
                        flag = false;
                }
        }
}
```

Fig. 3. Subtle DataRace Example

different execution. In particular, without any other dependencies but those in Figure 7, the property "e_1 must happen before e_3" can be violated by the program generating the execution in Figure 7, even though the particular observed run does not! Indeed, there is no evidence in the observed run that e_1 should precede e_3, because e_3 would happen anyway. Also, note that a standard, purely dynamic "happens-before" approach would *not* work in this example.

We implemented our approach in a prototype system called JPREDICTOR. The multi-threaded program to test is automatically instrumented to generate detailed execution traces and save them into log files. The log files are further filtered and finally analyzed by the tool. The prototype has been evaluated on several non-trivial applications and the results are promising. We were able not only to find known errors in large systems, but also reveal a wrong patch in the latest version of the Tomcat webserver.

More Motivating Examples

Below we first discuss the enhanced predictive ability of our technique by means of a more example, shown in Figure 3.

In this program, a mutex lock is implemented to allow the customizable light-weighted synchronization among threads. It provides two methods, namely, `tryAcquiring` to test and acquire the lock if possible and *release* to release the lock. *tryAcquiring* does *not* block the program; instead, it returns false if the acquiring fails, otherwise true is returned. This way, the thread can choose to wait for lock or continue to do other work first. This lock also counts the number of successful aquiring. However, this flexible implementation may cause tricky concurrent bugs if its behavior is not clear specified. In this example, *MyThread* tries to count the number of executed threads using the shared varible *MyThread.threadCount* that should be protected by the lock *MyThread.lock*. However, the implementation expects *tryAcquiring* to block the execution as a normal lock aquiring does. This misuse essentially causes a datarace of *MyThread.threadCount*.

It is not straightforward to accurately catch this bug using existing techniques. The lockset algorithm can detect this datarace, will also report dataraces of *lock.count* since the update is not protected by any lock, but in fact accesses to *lock.count* only occur when the lock is successfully aquired, which means that it is well protected from dataracing. For "happen-before" technique, since the execution of the thread finishes in a short time, it is very likely that the threads run in order without interleaving. In such case, there is a causal partial order from the previous release of the lock to the following aquiring due to the write and read on the variable *lock.flag*. Therefore, no datarace can be detected. (**CF:** once I make JMPaX work, we can give some more details here.)

If we take the dependence into consideration, we can see that the accesses to *count* in *tryAcquiring* depend on the previous condition checking of variable *flag*, since the if statement can choose to return from the method without update *count*. Therefore, the causal order caused by writes and reads of *flag* should be considered when detecting dataraces of *count*, which then enforces a total order among accesses to *count*. This way, our approach knows that there is no dataraces of *MyLock.count*. On the other hand, there is no dependence from the update of *MyThread.threadCount* to the checking of *lock.flag*, since not matter the value of *lock.flag* is, the update will be executed. So we can ignore

```
class Disk {
...
 public void  write(long  n, byte b) {
     disk.seek(n );
     disk.writeByte(b );
 }
…
}
```

```
class  Mutex  {
    boolean  locked=false;
    synchronized void   acq() {
      while (locked)   this.wait ();
      locked = true;
    }
    synchronized void   rel() {
      locked = false;
      this.notify ();
    }
}
```

Fig. 4. Implementation of Disk and Mutex locks in Daisy

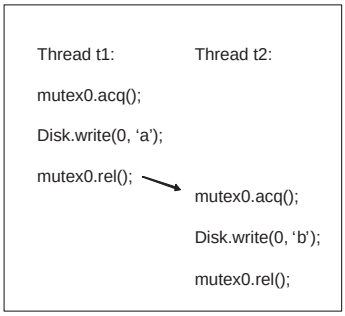

Fig. 5. Writes on a block

the accesses to *lock.flag* when we try to detect the datarace on *MyThread.threadCount*. Based on this observation, our approach shows that there are no causal relationship among those updates even when the threads are executed sequentially. The warning of potential dataraces is then reported.

Now let us look into a pratical example to show the complexity of reality. Daisy [23] is a small file system that was devised to serve as a concrete system to stimulate and challenge the various software verification tools. Figure 4 shows the write functions on disk and the implementation of Mutex locks in Daisy. The disk is simulated using the RandomAccessFile class in Java. The whole disk is logically divided into blocks, which are used to store files. Every block or file is protected by a specific Mutex lock for performance reasons.

Although efforts have been made to assure the well-defined synchronization of the disk operations, there is a data-race reported [23]. The data-race is caused by the usage of the *seek* function of the disk, which sets a pointer for the next file operation. Since the pointer is unique for the whole file, when two threads try to read/write different blocks, they will compete for the pointer without synchronization locks. Since the read/write operations on the disk are *not* protected by the system lock, the lock-set based algorithms will produce a large number of false alarms for disk accesses, overwhelming the actual data-race. In our approach, because the loop in the lock implementation is regarded as a non-terminating one, the subsequent accesses to the disk have a control-flow dependence on the loop condition. Therefore, accesses to the same disk block (a continuous area in the underlying array) are ordered by the reads/writes on the shared Mutex lock, as shown in Figure 5. So no races are reported for accesses to the array. But when two threads try to access different blocks, they use different Mutex locks and therefore a race on the file pointer will be reported.

Let us further consider the buggy implementation of Mutex in Figure 6. In this code, the while loop is replaced by an if statement, which will cause errors if multiple threads are waiting for the same lock. However, because of the causal partial order caused by read/write on the *locked* field, the potential data-race will not be detected by the traditional causal partial order based approaches. In our approach, because accesses to the shared memory, including the file pointer and the specific block in the disk, are out of the control scope of the if statement, there is no causal partial order between the two writes in Figure 5. Therefore races will be reported.

```
class Mutex {
...
    synchronized void acq() {
        if (locked) this.wait();
        locked = true;
    }
}
```

Fig. 6. Buggy lock acquiring

The point we are trying to make here is that one can significantly improve over existing work in predictive runtime analysis if one considers structural information about the program, obtained via static analysis of its source code. Observers of multi-threaded runs can generate other potential valid runs that can yield sequences of states that are complete only up-to-relevant events. This way observers can explore more potential interleavings, increasing their predictive power.

2 Parametric Framework for Control Dependence

Before we can technically discuss the dependence-sliced causality in our approach, we need to define the important concept of *control scope* of a statement. Briefly, the scope of a statement S is the set of statements whose reachability depends upon the potential choice at S. For example, in Figure 8 (A), the choice made at C_1 decides whether S_1 and S_2 are executed or not, but does not affect the execution of S_3. So $scope(C_1) = \{S_1, S_2\}$ and $scope(S_1) = scope(S_2) = \emptyset$.

This can be computed entirely statically by examining the structure of the program. One may also notice that the concept of control scope can be connected to those of control dependences ([13, 22]) which have been extensively studied in program analysis. In what follows we introduce a generic parametric framework, which can be instantiated to capture the existing control dependences and also a new control dependence, named termination-sensitive control dependence.

Control dependence plays a fundamental role in program analysis, e.g., in program slicing [15, 30], in compiler optimization [13, 1], in total program correctness [22], in security (of information flows) [11], etc. Intuitively, a statement S control depends on a choice statement C iff the choice made at C determines whether S is executed or not. Because of the significance and broad range of applications of control dependence, related definitions and algorithms have been extensively investigated: [13] gives an efficient algorithm to compute the control dependence; [22] introduces the strong control dependence, also called the range of the control statement in [33], as well as the weak control dependence; [3] defines a generalized notion of control dependence to capture both the classic and the weak direct control dependencies, together with their afferent algorithms.

Although all these notions of control dependence are related to each other, there is no adequate unifying framework for all of them, not even a uniform or consistent terminology. This often results in confusion and difficulty in understanding existing works, and may slow future developments in the area, in particular defining new, or domain-specific control dependence relations. For example, the strong control dependence in [22] is the transitive closure of the control dependence in [13], contradicting common sense in formal terminology, because the former is actually weaker than the latter; the generalized control dependence in [3] addresses only the *direct* control dependencies (classic and weak), but omits the word "direct" in definitions and proofs, and it also proposes the terminology "loop control dependence" for (direct) weak control dependence; [22] claims that the strong control dependence is included in the weak control dependence, which appears quite intuitive, but it is non-trivial to prove rigorously. We believe that a rigorous development of a unifying framework for the various control dependences would enhance understanding and terminology in this area.

A first important step in this direction is made in [3], where a generalized control dependence is defined, parametrized by a property on paths. It generalizes both the classic and the weak direct control dependences. A linear time algorithm is also given in [3], to detect all the statements that depend upon a given choice statement. However, the parametric approach in [3] covers only the *direct* control dependence. The first part of our work, on parametric control dependence (Section 2.3), can be regarded as an extension of the work in [3] to also include *indirect* control dependencies, as well as comparisons of different instances. Our compact prefix-invariance property of the parameter is equivalent to the intersection of all the constraints on the parameter in [3] required by the different results, though we do not need to add a self-looping edge in the terminal node of the control-flow graph in order to capture the weak control dependence. We also develop a rigorous mathematical theory in Section 2.3, capturing formally many of the "folklore" results about different control dependencies.

As an instance of the parametric framework, we define a new control dependence relation that we call *termination-sensitive control dependence*, because it is is sensitive to the termination information of loops, which can be given as annotations. If all loops are annotated as terminating then the termination-sensitive control dependence becomes the classic control dependence, while if all loops are annotated as non-terminating then it becomes the weak control dependence. Since in practice some loops are terminating and others are not, termination-sensitive control dependence is expected to improve the precision of analysis tools using it.

Our motivation for the termination-sensitive control dependence came from efforts in improving the accuracy and the coverage of predictive runtime analysis [7] of multithreaded systems. Since we refer back to it later in the paper, we explain this runtime analysis on a very simple example. Assume the threads and events in Figure 7, where e_1 causally precedes, or "happens-before", e_2 (e.g., e_1 writes a shared variable and e_2 reads it right afterwards), and the statement generating e'_3 is in the *control scope* (this notion will be formally defined in Section 3) of the statement generating e_2, while the statement generating e_3 is not in the control scope of e_2. Then we say that e'_3 *is dependent upon* e_1, but that e_3 is *not* dependent upon e_1, despite the fact that e_1 obviously happened before e_3. The intuition here is that e_3 would happen anyway, with or without e_1 happening. Because of its combined static/dynamic flavor, we call this new dependence relation on events the *hybrid dependence*. Interestingly, if the events in the scope of e_2 are not relevant for the property to check, then any permutation/linearization of relevant events consistent with the intra-thread total order and the hybrid dependence corresponds to some valid execution of

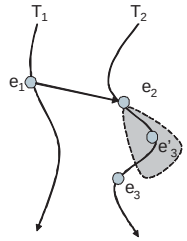

Fig. 7. Predictive Analysis

the multithreaded system. Therefore, if any of these permutations violate the property, then the system can do so in a different execution. In particular, without any other dependencies but those in Figure 7, the property "e_1 must happen before e_3" can be violated by the program generating the execution in Figure 7, even though the particular observed run does not! Indeed, there is no evidence in the observed run that e_1 should precede e_3, because e_3 would happen anyway.

The control scope of a statement is determined statically, as the set of statements that control depend on it. Unfortunately, the classic control dependence does not consider non-terminating loops, thus leading to false positives in the analysis, while the weak control dependence makes the worst case assumption (all loops are not terminating), resulting in over-restricted dependence among events and thus false negatives. The termination-sensitive control dependence takes the termination information of loops into account in order to build a more precise control dependence. We define this new control dependence as an instantiation of the parametric framework and show that it has properties similar to other control dependences. Interestingly, it follows as a corollary of the parametric framework that the termination-sensitive control dependence is stronger than the weak control dependence but weaker than the classic control dependence.

Finally, we describe an $O(|V|^2)$ algorithm to compute the control scopes in the context of higher level programming languages, such as Java and C#. The detailed explanation and proof of the algorithm is out of the scope this paper. Due to space limitations, proofs are all omitted in this paper. The interested reader is referred to [7] for detailed proofs, as well as for more details on predictive runtime analysis.

2.1 Preliminaries.

A *directed graph* G is a pair $\langle V, E\rangle$, where$E \subseteq V \times V$. The elements of V are called *nodes* and those of E are called *edges*. A *finite path* of G is a finite sequence of nodes $u_1u_2...u_{m+1}$ such that $(u_i, u_{i+1}) \in E$ for all $0 < i \leq m$, where $m > 0$ is its *length*. If $u = u_1$ and $v = u_{m+1}$ then we call this path a $u - v$ *path*. For any node u, we let λ_u be the empty path from u to itself; its length is 0. An *infinite path* is an infinite sequence $u_1u_2...$ such that $(u_i, u_{i+1}) \in E$ for all $i > 0$. A u*−path* is a (finite or infinite) path starting with u. We let *Paths*(G) be the set of all paths of G, finite or infinite. For a path π either infinite or finite of length larger than or equal to $k \geq 0$, we let $\pi|_k$ be the path containing the first k edges of π, i.e., $u_1u_2...u_{k+1}$. We also define the concatenation of paths: if $\alpha = u_1u_2...u_m$ finite and $\pi = u_mu_{m+1}...$ finite or infinite, then $\alpha\pi$ is the finite or infinite path $u_1u_2...u_mu_{m+1}....$ A *property* of paths in a graph G is a set $\mathcal{P} \subseteq$ *Paths*(G). For any $\pi \in$ *Paths*(G), we say that $\mathcal{P}(\pi)$ holds, or simply $\mathcal{P}(\pi)$, iff $\pi \in \mathcal{P}$.

Definition 1. *[13] A* ***control flow graph*** $CFG = \langle V, E, START, END\rangle$ *is a directed graph* $\langle V, E\rangle$*, together with an* ***entry node****, START, from which every other node is reachable, and an* ***exit node****, END, which is reachable from any other node. We make the standard assumption that nodes in V except END can have either one or two successors. Let* $V_C \subseteq V$ *denote the set of nodes with two successors and call them* ***choice nodes****.*

Intuitively, nodes in V correspond to statements in the program, edges in E indicate possible flows of control in the program, and choice nodes correspond to choice statements, such as conditionals, e.g., C_1 in Figure 8 (A). Conditionals can also be parts of loops, e.g., C_1 and C_2 in Figure 8 (C). Because of the assumption on the bounded number of successors, $|E| = O(|V|)$. Note that, in this paper, we tend to use letters at the beginning of the Greek alphabet, such as α, β, γ, etc., for $u - v$ paths, and letters π, π' and so on, for infinite or $u - END$ paths, though this convention is not strictly obeyed.

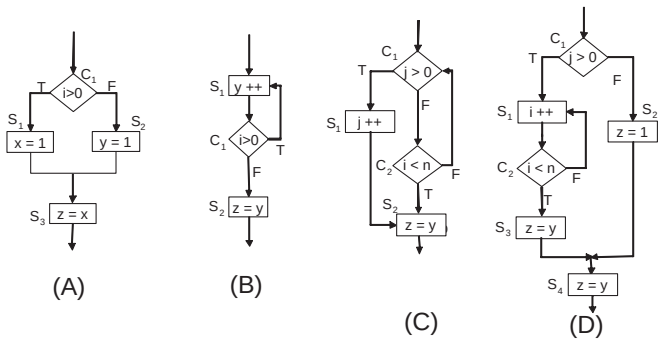

Fig. 8. Some control flow graphs

2.2 Control Dependence Revisited

There have been many studies on control dependence. We here discuss some of the major known results, introducing at the same time a uniform notation and terminology. Some of the results in this section are mentioned in other works as "folklore"; however, we were not able to find them proved formally in the literature; we will show that all these results follow as corollaries of the general results in the next section. The structure of the results in this section anticipates the structure of those for parametric control dependence in the next section. From here on we fix a CFG.

Classic Control Dependence

Definition 2. *([13,11]) Node u is **post-dominated** by node v, written $u \diamond\!\!\rightarrow v$, iff all $u - END$ paths contain v. Let **PostDom**(u) be the set of post-dominators of u except u.*

We use the notation $u \diamond\!\!\rightarrow v$ to symbolize the fact that no matter how we leave u (the first two edges of the diamond), we eventually converge (the other two edges of the diamond) and reach (the arrow) v. For example, in Figure 8 (A), $C_1 \diamond\!\!\rightarrow S_3$, while S_1 and S_2 do *not* post-dominate C_1; and in Figure 8 (B), $C_1 \diamond\!\!\rightarrow S_2$, while S_1 is *not* a post-dominator of C_1; however, note that in this figure there is no guarantee that S_2 will be reached once C_1 is reached, because of the potentially infinite loop through C_1. In our context of predictive runtime analysis, this reflects a serious limitation of the classic notion of control dependence; we will discuss this issue shortly.

Lemma 1. *The post-dominance relation, $\diamond\!\!\rightarrow$, is a partial order on the nodes of the CFG.*

Proof: The reflexivity is immediate. For transitivity, assume that $u \diamond\!\!\rightarrow v$ and $v \diamond\!\!\rightarrow w$. Then any $u - END$ path passes through v, and since any $v - END$ path passes through w, it follows that any $u - END$ path passes through w, that is, $u \diamond\!\!\rightarrow w$. For anti-symmetry, assume that $u \diamond\!\!\rightarrow v$ and $v \diamond\!\!\rightarrow u$, that is, that v belongs to any $u - END$ path and u belongs to any $v - END$ path. If $u \neq v$, then one can immediately see the contradiction, because only one of u or v can appear last on any finite path. Therefore, $u = v$. □

One can prove the following properties of post-dominance, while here they are just immediate corollaries of more general results on parametric control dependence in Section 2.3.

Corollary 1. *If $v_1 \neq v_2 \in PostDom(u)$ then either $v_1 \diamond\!\!\rightarrow v_2$ or $v_1 \diamond\!\!\rightarrow v_2$, i.e., $\langle PostDom(u), \diamond\!\!\rightarrow\rangle$ is a total order. As a consequence, for any u, if $PostDom(u) \neq \emptyset$ then $PostDom(u)$ has a unique first element w.r.t. $\diamond\!\!\rightarrow$.*

Proof: It follows by Definition 12, Lemma 4 and Proposition 3. □

Definition 3. *Let $ipd(u)$ be the first element of $\langle PostDom(u), \diamond\!\!\rightarrow\rangle$, called the **immediate post-dominator** of u; let $u \diamond\!\!\rightarrow v$ iff $v = ipd(u)$.*

The immediate post-dominator is the post-dominator that appears first on *any* $u - END$ path. For example, in Figure 8 (A), $C_1 \diamond\!\!\rightarrow S_3$ since S_3 appears before any other post-dominators of C_1 on any path from C_1 to *END*; in Figure 8 (B), $C_1 \diamond\!\!\rightarrow S_2$.

Proposition 1. *$\diamond\!\!\rightarrow$ is an inverted tree rooted by END.*

Therefore one can represent $\diamond\!\!\rightarrow$ as a *post-dominance tree* [20, 13] with *END* at its root. An $O(|V|\cdot\alpha(|V|))$ algorithm to compute such trees in given in [20], where α is the inverse Ackerman function. Based on post-dominance, *direct control dependence* can be defined as in [13]. Note that what we call "direct control dependence" below was called just "control dependence" in [13]; we believe, for reasons given shortly, that *its transitive closure* should be called control dependence.

Definition 4. *Node v is **directly control dependent** on node u, written $u \overset{dcd}{\rightsquigarrow} v$, iff*

1. *there exists a $u - v$ path α such that v post-dominates every node in α different from u; and*
2. *u is <u>not</u> post-dominated by v.*

For example, in Figure 8 (A), S_1 and S_2 are directly control dependent on C_1 but S_3 is not; while in Figure 8 (B), S_1 is directly control dependent on C_1 but S_2 is not. In Figure 8 (C), S_1 is directly control dependent on C_1 but not on C_2 (because S_1 does not post-dominate C_1). Note, however, that once C_2 is reached, *the execution of* S_1 *depends on the control decision made at* C_2! Therefore, S_1 *control depends* on C_2 by all practical, theoretical and intuitive means, suggesting that the terminology in [13] for control dependence is, perhaps, not the most appropriate one. We will shortly see that S_1 is in the *transitive closure* of the direct control dependence on C_2; for some reason, this transitive closure of the direct control dependence was misleadingly called "strong control dependence" in [22]. We will call it simply "control dependence" in what follows, because we think it captures best the *dependence* of some statements on the *control* decision made by others. Note that direct control dependence is *not* a partial order on nodes: in Figure 8 (C), C_1 and C_2 are directly control dependent on each other.

The notion of direct control dependence has been widely used in program analysis, e.g., in program slicing [15, 30] and compiler construction [13], where it was called control dependence . As we have already mentioned, we prefer to call it "direct", because it only considers *direct* dependence among statements; it does *not* reflect *indirect* dependence, e.g., the dependence from C_2 to S_1 in Figure 8 (C). Due to the hybrid dynamic/static setting of predictive runtime analysis, we need to also consider the "indirect" control dependence in the analysis. For example, in Figure 8 (C), if $C_1C_2C_1S_1S_2$ is an execution, the analysis needs to know that S_1 also depends on the choice made at C_2 to *not* exit the loop, which is caused by an indirect control dependence in the CFG.

In fact, even before the direct control dependence was introduced by Ferrante *et al.* [13], Dennings [11] already discussed the indirect influence of control statements on the program flow. Besides, Weiser [33] introduced a similar notion, called the *range* of branches, which is nothing but the transitive closure of the direct control dependence, as informally mentioned in [13, 24] without proof. Podgurski and Clarke [22] called it "strong control dependence", to emphasize that it was stronger than their "weak" dependence, still without proving that it was the transitive closure of the direct control dependence – otherwise, they would have probably noticed the inconsistent terminology: for a relation R, which is the control dependence with the terminology in [22], "strong R" ended up strictly including R. For reasons explained above, we prefer to drop the adjective "strong" and call it just control dependence:

Definition 5. *Node v is* ***control dependent*** *on u, written $u \overset{cd}{\rightsquigarrow} v$, iff there exists some $u - v$ path that does* not *contain $ipd(u)$, the immediate post-dominator of u.*

For example, in Figure 8 (C), $C_2 \overset{cd}{\rightsquigarrow} S_1$. One can prove the following properties of the control dependence, all of which following from our parametric framework:

Corollary 2. *For $\overset{dcd}{\rightsquigarrow}$ and $\overset{cd}{\rightsquigarrow}$, the following hold:*

1. *If $u \overset{dcd}{\rightsquigarrow} v$ then $PostDom(u) \subseteq PostDom(v)$; in particular, $ipd(v) \diamond\!\!\rightarrow ipd(u)$;*
2. *If $u \overset{cd}{\rightsquigarrow} v$ then $PostDom(u) \subseteq PostDom(v)$; in particular, $ipd(v) \diamond\!\!\rightarrow ipd(u)$;*
3. *$u \overset{cd}{\rightsquigarrow} v$ iff there exists some $u - v$ path α such that $\alpha \cap PostDom(u) = \emptyset$;*
4. *$\overset{dcd}{\rightsquigarrow} \subseteq \overset{cd}{\rightsquigarrow}$, that is, $u \overset{dcd}{\rightsquigarrow} v$ implies $u \overset{cd}{\rightsquigarrow} v$;*
5. *$\overset{cd}{\rightsquigarrow}$ is transitive, that is, $u \overset{cd}{\rightsquigarrow} v$ and $v \overset{cd}{\rightsquigarrow} w$ implies $u \overset{cd}{\rightsquigarrow} w$; and*
6. *$\overset{cd}{\rightsquigarrow} = \overset{dcd^+}{\rightsquigarrow}$, that is, $u \overset{cd}{\rightsquigarrow} v$ iff $u \overset{dcd^+}{\rightsquigarrow} v$.*

Proof:

1. It follows by Definition 12, Lemma 4 and Lemma 5.
2. It follows by Definition 12, Lemma 4 and Lemma 6.
3. It follows by Definition 12, Lemma 4 and Lemma 7.
4. It follows by Definition 12, Lemma 4 and Lemma 6 (1).
5. It follows by Definition 12, Lemma 4 and Lemma 6 (2).
6. It follows by Definition 12, Lemma 4 and Lemma 6 (3).

□

Therefore, control dependence is nothing but the transitive closure of the direct control dependence, so it is *weaker* than the direct control dependence. Driven by common sense in mathematical terminology ("stronger" means more

restrictive and "weaker" means more general), we therefore took the liberty and brevity to suggest what we believe is a more appropriate terminology for control dependencies than the one in [22].

Weak Control Dependence

Although control dependence as defined above captures the "indirect" dependence as well, it still has another important limitation: it is insensitive to (non-terminating) loops; e.g., in Figure 8 (C), S_2 is *not* control dependent on C_1 because the former is the post-dominator of the latter. This may lead to incorrect predictive analysis of multi-threaded systems. Re-consider the execution in Figure 7. Suppose it is generated by the program in Figure 8 (C). More specifically, suppose that e_1 is a write on the shared variable j, e_2 is the following read on j generated by C_1, e_3' is the write on j generated by S_1, and e_3 is the write on z generated by S_2. One may think that e_3 is *not* control dependent on e_2 by definition, that is, that e_3 will happen regardless of e_2. However, we can see that, since the loop is potentially non-terminating, S_2 may *never be executed* at runtime. Thus, the observed existence of e_3 is a consequence of a fortunate control choice made by C_1 when e_2 took place. So e_3 *should be control dependent* on e_2. Podgurski and Clarke [22] introduced strong post-dominance to handle control dependence in the presence of loops:

Definition 6. *Node u is **strongly post-dominated** by v, written $u \overset{s}{\diamond\!\!\rightarrow} v$, iff*

1. *$u \diamond\!\!\rightarrow v$ and*
2. *there is some integer $k \geq 1$ s.t. every $u-$path of length larger than or equal to k passes through v.*

*Node v is a **proper strong post-dominator** of u if $u \overset{s}{\diamond\!\!\rightarrow} v$ and $u \neq v$.*

In other words, u is strongly post-dominated by v iff u is post-dominated by v and there is no infinite $u-$path that does *not* pass through v; e.g, in Figure 8 (B), S_2 does not strongly post-dominate C_1, because there is an infinite path from C_1 that will not pass through S_2, while in Figure 8 (D), S_1 is strongly post-dominated by C_2 but C_2 is not strongly post-dominated by S_3. There may be no proper strong post-dominators for some nodes; e.g., in Figure 8 (C), neither C_1 nor C_2 have proper strong post-dominators, since they can choose to either stay in the loop forever or jump out of it. Based on strong post-dominance, weak control dependence is defined in [22] as follows:

Definition 7. *Node v is **directly weakly control dependent** on u, written $u \overset{dwcd}{\rightsquigarrow} v$, iff u has successors u' and u'' in the CFG such that $u' \overset{s}{\diamond\!\!\rightarrow} v$ but u'' is not strongly post-dominated by v; **weak control dependence**, written $\overset{wcd}{\rightsquigarrow}$, is the transitive closure of $\overset{dwcd}{\rightsquigarrow}$.*

In Figure 8 (D), $C_1 \overset{dwcd}{\rightsquigarrow} S_4$ because $S_2 \overset{s}{\diamond\!\!\rightarrow} S_4$ but not $S_1 \overset{s}{\diamond\!\!\rightarrow} S_4$. Weak control dependence is a generalization of control dependence, that is, every control dependence is a weak control dependence. This was informally mentioned in [22], but it is not straightforward to prove rigorously using their original definitions. However, it will follow as a corollary of more general results in our parametric framework, as shown at the end of Section 2.3. What makes this result even more interesting is that *direct* weak control dependence is *not* a generalization of *direct* control dependence. E.g., in Figure 8 (D), S_3 is directly control dependent but not directly weak control dependent on C_1, while it is directly weak control dependent but not directly control dependent on C_2. This may suggest that the terminology "direct weak control dependence" versus "direct control dependence" is also inappropriate, because the former is not weaker than the latter. However, this is not not problematic here because the adjective "weak" does not qualify the relation "direct control dependence", but the relation "control dependence"; the terminology "weak direct control dependence" would have been indeed problematic. Like control dependence, weak control dependence is not a partial order either: e.g., in Figure 8 (C), both $C_1 \overset{dwcd}{\rightsquigarrow} C_2$ and $C_2 \overset{dwcd}{\rightsquigarrow} C_1$.

The (direct) weak control dependence makes the worst-case assumption that all loops are non-terminating, which is very rarely the case in practice. In fact, most loops in real programs do *terminate*. We next propose a parametric framework to define and reason about control dependence, which incorporates both the direct control dependence and the direct weak control dependence, as well as their transitive closures, as special cases. This framework can be easily instantiated to define other control dependence relations, such as the termination-sensitive control dependence discussed in Section 2.4 that we need for predictive runtime analysis.

2.3 Parametric Control Dependence

Definition 8. *A set $\mathcal{P} \subseteq Paths(CFG)$ is a **prefix-invariant property** on paths iff*

1. $\mathcal{P}(\lambda_{END})$*; and*
2. $\mathcal{P}(\alpha\pi) \Leftrightarrow \mathcal{P}(\pi)$ *for any* $\alpha\pi \in Paths(CFG)$ *(*α *is obviously finite).*

From now on in this section, we fix a prefix-invariant property $\mathcal{P}$. One can show that $\mathcal{P}$ contains all $u-END$ paths, that is, that $\mathcal{P}(\alpha)$ holds for any $u-END$ path α.

Definition 9. *A* $u \overset{\mathcal{P}}{-}$ *path is any u–path in* $\mathcal{P}$*.*

For any node u, there exists at least one finite $u \overset{\mathcal{P}}{-}$ path (*END* is reachable from u).

Definition 10. *Node u is* $\mathcal{P}$***-post-dominated*** *by node v, written* $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$*, iff all* $u \overset{\mathcal{P}}{-}$*paths contain v. Let* $\boldsymbol{PostDom_{\mathcal{P}}}(u)$ *denote the set of* $\mathcal{P}$*-post-dominators of u different from u.*

Note that for some nodes u, $PostDom_{\mathcal{P}}(u)$ can be empty. For example, as shown after Definition 6, some nodes may not have strong post-dominators, which will be proved shortly to be a special case of $\mathcal{P}$-post-dominators for a well chosen property $\mathcal{P}$. The following says that $\mathcal{P}$-post-dominance is stronger than classical post-dominance:

Proposition 2. $\overset{\mathcal{P}}{\diamond\!\!\rightarrow} \subseteq \diamond\!\!\rightarrow$*, that is,* $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$ *implies* $u \diamond\!\!\rightarrow v$*.*

Proof: Suppose that $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$. Since any (finite) $u-END$ path is a $u \overset{\mathcal{P}}{-}$ path (by Definition 9), it follows that any $u-END$ path contains v. Therefore, $u \diamond\!\!\rightarrow v$. □

Lemma 2. $\overset{\mathcal{P}}{\diamond\!\!\rightarrow}$ *is a partial order.*

Proof: The reflexivity is immediate. For transitivity, assume that $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$ and $v \overset{\mathcal{P}}{\diamond\!\!\rightarrow} w$. Then any $u \overset{\mathcal{P}}{-}$ path passes through v. Since $\mathcal{P}$ is prefix-invariant and any $v \overset{\mathcal{P}}{-}$ path passes through w, it follows that any $u \overset{\mathcal{P}}{-}$ path passes through w, that is, $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} w$. For anti-symmetry, assume that $v \overset{\mathcal{P}}{\diamond\!\!\rightarrow} u$ and $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$. Then we have $v \diamond\!\!\rightarrow u$ and $u \diamond\!\!\rightarrow v$ by Proposition 2, so $u = v$ by the anti-symmetry of $\diamond\!\!\rightarrow$ (Lemma 1). □

Lemma 3. *If* $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$ *and there is a* $u - u'$ *path that does not contain v, then* $u' \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$*.*

Proof: Suppose that $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$ and α is a u–u' path that does not contain v. Let π be a $u' \overset{\mathcal{P}}{-}$path. Since $\mathcal{P}$ is prefix-invairant, $\alpha\pi$ is a $u \overset{\mathcal{P}}{-}$path. Therefore, $v \in \alpha\pi$, that is, $v \in \pi$. □

Proposition 3. *If* $v_1 \neq v_2 \in PostDom_{\mathcal{P}}(u)$*, then either* $v_1 \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v_2$ *or* $v_2 \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v_1$*; in other words,* $\langle PostDom_{\mathcal{P}}(u), \overset{\mathcal{P}}{\diamond\!\!\rightarrow}\rangle$ *is a total order. As a consequence, if* $PostDom_{\mathcal{P}}(u) \neq \emptyset$ *then* $PostDom_{\mathcal{P}}(u)$ *has a unique first element w.r.t.* $\overset{\mathcal{P}}{\diamond\!\!\rightarrow}$*.*

Proof: As mentioned, there exists at least one $u \overset{\mathcal{P}}{-}$ path. For a $u \overset{\mathcal{P}}{-}$ path π, since $v_1, v_2 \in PostDom_{\mathcal{P}}(u)$, π contains both v_1 and v_2. Suppose that v_1 appears before v_2 on π, that is π has the form $\alpha_1 v_1 \alpha_2$, where α_1 is a $u - v_1$ path that does not contain v_2. Then $v_1 \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v_2$ by Lemma 3. If v_2 appears before v_1 on π then one can similarly show that $v_2 \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v_1$. □

Definition 11. *Let* $ipd_{\mathcal{P}}(u)$ *be the first element of the total order* $\langle PostDom_{\mathcal{P}}(u), \overset{\mathcal{P}}{\diamond\!\!\rightarrow}\rangle$*, called the* ***immediate*** $\mathcal{P}$***-post-dominator*** *of u; let* $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$ *iff* $v = ipd_{\mathcal{P}}(u)$*.*

Proposition 4. $\overset{\mathcal{P}}{\diamond\!\!\rightarrow}$ *is a forest of inverted trees.*

Proof: According to Lemma 3, for any node u with $PostDom_{\mathcal{P}}(u) \neq \emptyset$, u has only one successor w.r.t. $\diamond\!\!\rightarrow$, namely $ipd_{\mathcal{P}}(u)$. Therefore, each node in the CFG has at most one successor w.r.t. $\overset{\mathcal{P}}{\diamond\!\!\rightarrow}$. □

One can show that post-dominance and strong post-dominance are two special cases of $\mathcal{P}$-post-dominance by choosing appropriate parameters $\mathcal{P}$:

Definition 12. *Let $\mathcal{P}_\perp$ denote the set of all finite paths ending with END and let $\mathcal{P}_{\perp\infty}$ be the union of $\mathcal{P}_\perp$ with all infinite paths.*

Lemma 4. *Both $\mathcal{P}_\perp$ and $\mathcal{P}_{\perp\infty}$ are prefix-invariant.*

Proof: Both $\mathcal{P}_\perp$ and $\mathcal{P}_{\perp\infty}$ contain λ_{END}. $\mathcal{P}_\perp$ is clearly prefix-invariant because, for any $u - v$ path α, $\alpha\pi$ is a $u - END$ path if and only if π is a $v - END$ path. Also, $\mathcal{P}_{\perp\infty}$ is prefix-invariant because, for any $u - v$ path α, $\alpha\pi$ is a $u - END$ path or an infinite path if and only if π is a $v - END$ path or an infinite path. □

Proposition 5. $\diamond\!\!\rightarrow = \overset{\mathcal{P}_\perp}{\diamond\!\!\rightarrow}$ *and* $\overset{s}{\diamond\!\!\rightarrow} = \overset{\mathcal{P}_{\perp\infty}}{\diamond\!\!\rightarrow}$.

Proof: $\diamond\!\!\rightarrow = \overset{\mathcal{P}_\perp}{\diamond\!\!\rightarrow}$ follows by Definition 2, Definition 10 and Definition 12. For $\overset{s}{\diamond\!\!\rightarrow} = \overset{\mathcal{P}_{\perp\infty}}{\diamond\!\!\rightarrow}$, suppose first that $u \overset{s}{\diamond\!\!\rightarrow} v$ and consider a $u \overset{\mathcal{P}_{\perp\infty}}{-}$ path π. If π is finite, i.e., a $u - END$ path, then $v \in \pi$ because $u \diamond\!\!\rightarrow v$ by Definition 6. If π is infinite, then there is some $k \geq 1$ such that $v \in \pi|_k$, so $v \in \pi$. Therefore, $u \overset{\mathcal{P}_{\perp\infty}}{\diamond\!\!\rightarrow} v$. Conversely, suppose $u \overset{\mathcal{P}_{\perp\infty}}{\diamond\!\!\rightarrow} v$. In particular, this means that any $u - END$ path contains v, so $u \diamond\!\!\rightarrow v$. Now suppose that there is no $k \geq 1$ such that $v \in \pi$ for any finite $u-$path π with $|\pi| \geq k$. In other words, for any $k \geq 1$, either there is no path longer than or equal to k or there is some path π such that $|\pi| \geq k$ and $v \notin \pi$. The first case means that there are only finite $u-$paths, in which case $\diamond\!\!\rightarrow$ and $\overset{s}{\diamond\!\!\rightarrow}$ coincide, so $u \overset{s}{\diamond\!\!\rightarrow} v$. For the second case, since the CFG has a finite number of nodes, one can choose a large enough k such that, by the pidgeon-hole principle, any finite $u-$path π must contain a duplicate of some node w when $|\pi| \geq k$. So we can have such a π in the form of $\alpha w\beta w\gamma$ and $v \notin \pi$. We can then build an infinite $u-$path $\alpha(w\beta)^\infty$ which does not contain v. This contradicts the hypothesis. □

We will discuss a third special case of $\mathcal{P}$-post-dominance in Section 2.4, where additional termination information of loops will be taken into account.

Definition 13. *v is **directly $\mathcal{P}$-control dependent** on u, written $u \overset{d\mathcal{P}}{\rightsquigarrow} v$, iff*

1. *there exists some $u - v$ path α such that v $\mathcal{P}$-post-dominates all nodes in α except u, and*
2. *v does <u>not</u> $\mathcal{P}$-post-dominate u.*

Note that $\overset{d\mathcal{P}}{\rightsquigarrow}$ is *not* a partial order. For example, $\overset{dcd}{\rightsquigarrow}$ and $\overset{dwcd}{\rightsquigarrow}$, which will be shortly proved to be special cases of $\overset{d\mathcal{P}}{\rightsquigarrow}$, are not partial orders. This means that, in the worst case, the time needed to compute the transitive closure of $\overset{d\mathcal{P}}{\rightsquigarrow}$ using the standard transitive closure algorithms is $O(|V|^3)$ [10]. In Section 3 we give an alternative $O(|V|^2)$ algorithm that works on more restrictive CFGs, such as those obtained from programs in modern languages, e.g., Java and C#.

Lemma 5. *If $u \overset{d\mathcal{P}}{\rightsquigarrow} v$ then $PostDom_{\mathcal{P}}(u) \subseteq PostDom_{\mathcal{P}}(v)$; hence, $ipd_{\mathcal{P}}(v) \overset{\mathcal{P}}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}}(u)$.*

Proof: By Definition 13, there exists a $u - v$ path α, such that v $\mathcal{P}$-post-dominates any node in α except u. For any node $u' \in PostDom_{\mathcal{P}}(u)$, u' cannot belong to α; otherwise, $u' \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$, because v $\mathcal{P}$-post-dominates all nodes on α except u, and thus $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$ by Lemma 2, which contradicts $u \overset{d\mathcal{P}}{\rightsquigarrow} v$. Suppose, by contradiction, that u' does not $\mathcal{P}$-post-dominate v; then there exists a $v \overset{\mathcal{P}}{-}$path π that does not contain u'. Therefore, we can build a $u \overset{\mathcal{P}}{-}$path, namely $\alpha\pi$, that does not contain u', contradicting the fact that $u' \in PostDom_{\mathcal{P}}(u)$. Hence $PostDom_{\mathcal{P}}(u) \subseteq PostDom_{\mathcal{P}}(v)$. Then $ipd_{\mathcal{P}}(u) \in PostDom_{\mathcal{P}}(v)$, so $ipd_{\mathcal{P}}(v) \overset{\mathcal{P}}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}}(u)$. □

Definition 14. *Node v is **$\mathcal{P}$-control dependent** on u, written $u \overset{\mathcal{P}}{\rightsquigarrow} v$, iff there exists some $u - v$ path that does not contain $ipd_{\mathcal{P}}(u)$.*

Lemma 6. *If $u \overset{\mathcal{P}}{\rightsquigarrow} v$ then $PostDom_{\mathcal{P}}(u) \subseteq PostDom_{\mathcal{P}}(v)$; hence, $ipd_{\mathcal{P}}(v) \overset{\mathcal{P}}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}}(u)$.*

Proof: We first prove that $ipd_{\mathcal{P}}(u) \in PostDom_{\mathcal{P}}(v)$. By Definition 14, there exists a $u - v$ path α that does not contain $ipd_{\mathcal{P}}(u)$. If $ipd_{\mathcal{P}}(u)$ does not $\mathcal{P}$-post-dominate v, then there exits a $v \overset{\mathcal{P}}{-}$path π that does not contain $ipd_{\mathcal{P}}(u)$. Therefore, we can build a $u \overset{\mathcal{P}}{-}$path, namely $\alpha\pi$, that does not contain $ipd_{\mathcal{P}}(u)$, contradicting the definition of $ipd_{\mathcal{P}}(u)$. Therefore $ipd_{\mathcal{P}}(u) \in PostDom_{\mathcal{P}}(v)$. For any node $u' \in PostDom_{\mathcal{P}}(u)$, $ipd_{\mathcal{P}}(u) \overset{\mathcal{P}}{\diamond\!\!\rightarrow} u'$; therefore, by Lemma 2, $v \overset{\mathcal{P}}{\diamond\!\!\rightarrow} u'$. □

Lemma 7. *$u \overset{\mathcal{P}}{\rightsquigarrow} v$ iff there exists some $u - v$ path α such that $\alpha \cap PostDom_{\mathcal{P}}(u) = \emptyset$.*

Proof: It suffices to show that if α is a $u - v$ path that does not contain $ipd_{\mathcal{P}}(u)$ then α does not contain any $\mathcal{P}$-post-dominator of u. Suppose, by contradiction, that α does contain some proper $\mathcal{P}$-post-dominator u' of u different from $ipd_{\mathcal{P}}(u)$, that is, that α has the form $\alpha_1 u' \alpha_2$, where α_1 does not contain $ipd_{\mathcal{P}}(u)$. Since $ipd_{\mathcal{P}}(u)$ does *not* $\mathcal{P}$-post-dominate u' (otherwise $u' = ipd_{\mathcal{P}}(u)$ by Lemma 2), there is some $u' \overset{\mathcal{P}}{-}$path π that does not contain $ipd_{\mathcal{P}}(u)$. Since $ipd_{\mathcal{P}}(u) \notin \alpha_1$, it follows that $ipd_{\mathcal{P}}(u) \notin \alpha_1\pi$, contradiction. □

Proposition 6. *The following hold:*

1. $\overset{d\mathcal{P}}{\rightsquigarrow} \subseteq \overset{\mathcal{P}}{\rightsquigarrow}$;
2. $\overset{\mathcal{P}}{\rightsquigarrow}$ *is transitive; and*
3. $\overset{\mathcal{P}}{\rightsquigarrow} = \overset{d\mathcal{P}^+}{\rightsquigarrow}$.

Proof:

1. Suppose that $u \overset{d\mathcal{P}}{\rightsquigarrow} v$. In other words, there exists a $u - v$ path α such that v $\mathcal{P}$-post-dominates all nodes in α except u. Then $ipd_{\mathcal{P}}(u)$ cannot appear in α since otherwise $ipd_{\mathcal{P}}(u) \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$, implying that $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$ by Lemma 2, which contradicts the definition of $\overset{d\mathcal{P}}{\rightsquigarrow}$. Therefore $u \overset{\mathcal{P}}{\rightsquigarrow} v$.
2. Suppose that $u \overset{\mathcal{P}}{\rightsquigarrow} v$ and $v \overset{\mathcal{P}}{\rightsquigarrow} w$. Then there exists a $u - v$ path α that does not contain $ipd_{\mathcal{P}}(u)$ and a $v - w$ path β that does not contain $ipd_{\mathcal{P}}(v)$. By Lemma 6, $ipd_{\mathcal{P}}(v) \overset{\mathcal{P}}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}}(u)$. If $ipd_{\mathcal{P}}(u) = ipd_{\mathcal{P}}(v)$, then $ipd_{\mathcal{P}}(u)$ cannot appear in β. If $ipd_{\mathcal{P}}(u) \neq ipd_{\mathcal{P}}(v)$, then according to Lemma 2, $ipd_{\mathcal{P}}(v)$ does not post-dominate $ipd_{\mathcal{P}}(u)$. Thus, there exists an $ipd_{\mathcal{P}}(u) \overset{\mathcal{P}}{-}$path π that does not contain $ipd_{\mathcal{P}}(v)$. Suppose that $ipd_{\mathcal{P}}(u)$ appears in β, that is, that β has the form $\beta_1 ipd_{\mathcal{P}}(u)\beta_2$. Then we can build a $v \overset{\mathcal{P}}{-}$path $\beta_1\pi$ that does not contain $ipd_{\mathcal{P}}(v)$, contradicting the definition of $ipd_{\mathcal{P}}(v)$. So $ipd_{\mathcal{P}}(u)$ cannot appear in β. Therefore, we have found a $u - w$ path $\alpha\beta$ that does not contain $ipd_{\mathcal{P}}(u)$, that is, $u \overset{\mathcal{P}}{\rightsquigarrow} w$.
3. The first two items imply immediately that $\overset{d\mathcal{P}^+}{\rightsquigarrow} \subseteq \overset{\mathcal{P}}{\rightsquigarrow}$. For the other implication, suppose that $u \overset{\mathcal{P}}{\rightsquigarrow} v$ and let α be a $u - v$ path such that $ipd_{\mathcal{P}}(u) \notin \alpha$. We prove by well-founded induction on the length of α that $u \overset{d\mathcal{P}^+}{\rightsquigarrow} v$. Let w be the last node on α which is *not* $\mathcal{P}$-post-dominated by v. By Definition 13, it follows that $w \overset{d\mathcal{P}}{\rightsquigarrow} v$. If $w = u$ then we are done. If $w \neq u$ then $u \overset{d\mathcal{P}^+}{\rightsquigarrow} w$ by the induction hypothesis, so $u \overset{d\mathcal{P}^+}{\rightsquigarrow} v$.

□

One can also show that direct control dependence and direct weak control dependence are two special cases of direct $\mathcal{P}$-control dependence, while control dependence and weak control dependence are two special cases of $\mathcal{P}$-control dependence:

Proposition 7. *$\overset{dcd}{\rightsquigarrow} = \overset{d\mathcal{P}_\perp}{\rightsquigarrow}$ and $\overset{dwcd}{\rightsquigarrow} = \overset{d\mathcal{P}_{\perp\infty}}{\rightsquigarrow}$.*

Proof: $\overset{dcd}{\rightsquigarrow} = \overset{d\mathcal{P}_\perp}{\rightsquigarrow}$ follows by Definition 4, Definition 13, and Proposition 5. For $\overset{dwcd}{\rightsquigarrow} = \overset{d\mathcal{P}_{\perp\infty}}{\rightsquigarrow}$, since by Proposition 5, $\overset{s}{\diamond\!\!\rightarrow} = \overset{\mathcal{P}_{\perp\infty}}{\diamond\!\!\rightarrow}$, we use only $\overset{s}{\diamond\!\!\rightarrow}$ in this proof. Suppose that $u \overset{dwcd}{\rightsquigarrow} v$. Then u has two successors $u', u"$, such that $u' \overset{s}{\diamond\!\!\rightarrow} v$ and $u"$ is not strongly post-dominated by v. The latter implies that u is not strongly post-dominated by v. The former first implies that there is some $u - v$ path that does not contain v except at its end, and then, by Lemma 3, that v

$\mathcal{P}$-post-dominates all nodes on that path. Therefore, $u \overset{d\mathcal{P}_{\perp\infty}}{\rightsquigarrow} v$. Conversely, suppose $u \overset{d\mathcal{P}_{\perp\infty}}{\rightsquigarrow} v$. Then there exists a $u - v$ path α such that v strongly post-dominates all nodes in α except u, and v does not strongly post-dominate u. Let u' be the successor of u in α. Obviously, $u' \overset{s}{\diamond\!\!\rightarrow} v$. Besides, there exists a $u \overset{\mathcal{P}_{\perp\infty}}{-}$path $u\pi$ that does not contain v. Let u" be the successor of u in $u\pi$. Then we can have a u" $\overset{\mathcal{P}_{\perp\infty}}{-}$path, namely π, that does not contain v, that is to say, u" is not strongly post-dominated by v. Therefore, $u \overset{dwcd}{\rightsquigarrow} v$. □

Proposition 8. $\overset{cd}{\rightsquigarrow} = \overset{\mathcal{P}_{\perp}}{\rightsquigarrow}$ *and* $\overset{wcd}{\rightsquigarrow} = \overset{\mathcal{P}_{\perp\infty}}{\rightsquigarrow}$.

Proof: $\overset{cd}{\rightsquigarrow} = \overset{\mathcal{P}_{\perp}}{\rightsquigarrow}$ follows by Definition 5, Definition 14, and Proposition 5. $\overset{wcd}{\rightsquigarrow} = \overset{\mathcal{P}_{\perp\infty}}{\rightsquigarrow}$ is the immediate result of Proposition 6 and Proposition 7. □

The following proposition will allow us to *compare* control dependencies, based on just a simple comparison of their corresponding parameters:

Proposition 9. *If $\mathcal{P} \subseteq \mathcal{P}'$ are prefix-invariant properties then:*

1. $\overset{\mathcal{P}'}{\diamond\!\!\rightarrow} \subseteq \overset{\mathcal{P}}{\diamond\!\!\rightarrow}$;
2. $PostDom_{\mathcal{P}'}(u) \subseteq PostDom_{\mathcal{P}}(u)$;
3. $ipd_{\mathcal{P}}(u) \overset{\mathcal{P}}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}'}(u)$; *and*
4. $\overset{\mathcal{P}}{\rightsquigarrow} \subseteq \overset{\mathcal{P}'}{\rightsquigarrow}$.

Proof:

1. If $u \overset{\mathcal{P}'}{\diamond\!\!\rightarrow} v$ then all $u \overset{\mathcal{P}'}{-}$paths contains v. Since $\mathcal{P} \subseteq \mathcal{P}'$, all $u \overset{\mathcal{P}}{-}$paths are $u \overset{\mathcal{P}'}{-}$paths. Then all $u \overset{\mathcal{P}}{-}$paths contain v, that is, $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$.
2. For any $v \in PostDom_{\mathcal{P}'}(u)$, that is, $u \overset{\mathcal{P}'}{\diamond\!\!\rightarrow} v$, by the first item, $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} v$, that is, $v \in PostDom_{\mathcal{P}}(u)$.
3. By the first item, $u \overset{\mathcal{P}}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}'}(u)$. By Definition 11, $ipd_{\mathcal{P}}(u) \overset{\mathcal{P}}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}'}(u)$.
4. By Lemma 7, we only need to prove that, for a $u - v$ path α, if $\alpha \cap PostDom_{\mathcal{P}}(u) = \emptyset$ then $\alpha \cap PostDom_{\mathcal{P}'}(u) = \emptyset$. This follows by the second item.

□

Corollary 3. $\overset{cd}{\rightsquigarrow} \subseteq \overset{\mathcal{P}}{\rightsquigarrow}$ *for any prefix-invariant property $\mathcal{P}$; in particular,* $\overset{cd}{\rightsquigarrow} \subseteq \overset{wcd}{\rightsquigarrow}$.

Proof: Since every finite path ending with *END* is a $\mathcal{P}$ path, $\mathcal{P}_{\perp} \subseteq \mathcal{P}$. By Proposition 9 and Proposition 8, $\overset{cd}{\rightsquigarrow} \subseteq \overset{\mathcal{P}}{\rightsquigarrow}$, and in particular $\overset{cd}{\rightsquigarrow} \subseteq \overset{wcd}{\rightsquigarrow}$. □

Interestingly, the inclusion of the direct versions of the dependences in the corollary above does *not* hold. For example, it is *not* the case that $\overset{dcd}{\rightsquigarrow} \subseteq \overset{dwcd}{\rightsquigarrow}$ (see the discussion following Definition 7).

2.4 Termination-Sensitive Control Dependence

Weak control dependence takes loops into account using strong post-dominance, which is more suitable for proving total correctness of programs [22] than the classic control dependence. However, weak control dependence unfortunately makes the worst-case assumption about the termination of loops in the program, namely, all loops are assumed to be potentially infinite. Considering the fact that *most loops terminate* in real programs, this assumption is too conservative in practice. Let us look at the example in Figure 8 (D). The loop containing S_1 and C_2 obviously terminates, so S_3 will be eventually executed once C_2 is reached. In other words, the execution of S_3 *does not depend* on the choice made at C_2. However, by Definition 7, $C_2 \overset{wcd}{\rightsquigarrow} S_3$. Such over-restrictive assumptions may bring *false positives* to static program analysis, while for our runtime predictive analysis, they may generate over-restrictive control dependences on events,

reducing the number of potential permutations of events when investigating possible actual executions, resulting in more *false negatives*, i.e., a reduced coverage.

In this section, we introduce a new control dependence relation, named *termination-sensitive control dependence*, as another instantiation of the parametric control dependence framework presented in Section 2.3. As indicated by its name, this control dependence takes the termination information of loops into account in order to improve the precision of program analyses that make use of control dependence. Although termination analysis is an undecidable problem, there exist some effective algorithms to approximately determine termination of programs, e.g., [9, 4] (more discussion on these algorithms is out of the scope of this paper). Besides, termination information can also be provided by users (e.g., using special annotations) or detected by heuristics-based criteria (for example, a loop whose condition is $i < n$ and in which i is increased at each iteration will always terminate). Here we only focus on defining a more precise control dependence relation using existing termination information, which is assumed to be correct.

First, we extend the CFG with termination information:

Definition 15. *A **termination-sensitive control flow graph** $\langle V, E, START, END, V_\infty\rangle$ is a CFG $\langle V, E, START, END\rangle$ together with a distinguished set of nodes $V_\infty \subseteq V$.*

The nodes in V_∞ can be thought of as nodes that can lead to non-terminating executions. In practice, one would like to annotate as few statements as possible to provide the termination information; if that is the case, then V_∞ can contain precisely the conditions of those loops that may not terminate in some executions. Theoretically, one can add to V_∞ *all* the unavoidable statements in such loops, but this is not necessary. Besides, some of these statements can be themselves loops, but ones which terminate. From here on, we fix an arbitrary termination-sensitive CFG and define complete paths as follows:

Definition 16. *A **complete path** π is a path either finite and ends with END, or infinite and $inf(\pi) \cap V_\infty \neq \emptyset$, where $inf(\pi)$ gives those nodes visited infinitely often in π. Let $\mathcal{P}_\top$ denote the set of complete paths of the termination-sensitive CFG.*

Note that infinite paths generated by "nested" loops in which the outer ones are annotated as "non-terminating" (in V_∞), while the inner ones are "terminating", are considered complete as far as the outer loop is executed infinitely often. One may be tempted to instead annotate the "teminating" nodes as a subset $V_\top \subseteq V$ and then require the complete path to satisfy $inf(\pi) \cap V_\top = \emptyset$; however, such an approach would be less precise, because it would exclude common paths as the ones generated by nested loops as above. There is an interesting similarity between termination-sensitive CFG and Buchi automata [5], where the role of *accepting states* is played by V_∞ and that of *accepted words* by complete paths.

One can show that $\mathcal{P}_\top$ is also a prefix-invariant property on paths. Indeed, for any $u - v$ path α and $v-$path π, $\alpha\pi$ is a $u - END$ path iff π is a $v - END$ path. Besides, if $\alpha\pi$ is infinite, then since α is finite, $inf(\alpha\pi) = inf(\pi)$. Therefore, $inf(\alpha\pi) \cap V_\infty = inf(\pi) \cap V_\infty$; in particular, $inf(\alpha\pi) \cap V_\infty \neq \emptyset$ iff $inf(\pi) \cap V_\infty \neq \emptyset$. Based on the parametric framework for control dependence introduced in Section 2.3, we can define corresponding post-dominance and dependence notions: $\mathcal{P}_\top$-post-dominance ($\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}$), immediate $\mathcal{P}_\top$-post-dominance ($\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}$), direct $\mathcal{P}_\top$-control dependence ($\overset{d\mathcal{P}_\top}{\rightsquigarrow}$), and $\mathcal{P}_\top$-control dependence ($\overset{\mathcal{P}_\top}{\rightsquigarrow}$). The following results follow immediately from the generic framework in the previous section:

Corollary 4. *For $\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}$, the following hold:*

1. *$\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} \subseteq \diamond\!\!\rightarrow$, that is, $u \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} v$ implies $u \diamond\!\!\rightarrow v$;*
2. *$\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}$ is a partial order;*
3. *If $v_1 \neq v_2 \in PostDom_{\mathcal{P}_\top}(u)$, then either $v_1 \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} v_2$ or $v_2 \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} v_1$; in other words, $\langle PostDom_{\mathcal{P}_\top}(u), \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}\rangle$ is a total order;*
4. *If $PostDom_{\mathcal{P}_\top}(u) \neq \emptyset$ then $PostDom_{\mathcal{P}_\top}(u)$ has a unique first element w.r.t. $\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}$;*
5. *$\overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow}$ is a forest of inverted trees;*

Proof:

1. It follows by Proposition 2.
2. It follows by Lemma 2.

3. It follows by Proposition 3.
4. It follows by Proposition 3.
5. It follows by Proposition 4.

□

Corollary 5. *For $\overset{d\mathcal{P}_\top}{\rightsquigarrow}$ and $\overset{\mathcal{P}_\top}{\rightsquigarrow}$, the following hold:*

1. *If $u \overset{d\mathcal{P}_\top}{\rightsquigarrow} v$ then $PostDom_{\mathcal{P}_\top}(u) \subseteq PostDom_{\mathcal{P}_\top}(v)$; in particular, $ipd_{\mathcal{P}_\top}(v) \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}_\top}(u)$;*
2. *If $u \overset{\mathcal{P}_\top}{\rightsquigarrow} v$ then $PostDom_{\mathcal{P}_\top}(u) \subseteq PostDom_{\mathcal{P}_\top}(v)$; in particular, $ipd_{\mathcal{P}_\top}(v) \overset{\mathcal{P}_\top}{\diamond\!\!\rightarrow} ipd_{\mathcal{P}_\top}(u)$;*
3. *$u \overset{\mathcal{P}_\top}{\rightsquigarrow} v$ iff there exists some $u - v$ path α such that $\alpha \cap PostDom_{\mathcal{P}_\top}(u) = \emptyset$;*
4. *$\overset{d\mathcal{P}_\top}{\rightsquigarrow} \subseteq \overset{\mathcal{P}_\top}{\rightsquigarrow}$;*
5. *$\overset{\mathcal{P}_\top}{\rightsquigarrow}$ is transitive; and*
6. *$\overset{\mathcal{P}_\top}{\rightsquigarrow} = \overset{d\mathcal{P}_\top}{\rightsquigarrow}^+$.*

Proof:

1. It follows by Lemma 5.
2. It follows by Lemma 6.
3. It follows by Lemma 7.
4. It follows Lemma 6 (1).
5. It follows Lemma 6 (2).
6. It follows Lemma 6 (3).

□

Now we are ready to define the termination-sensitive control dependence and to compare this new control dependence with the classical and weak control dependence:

Definition 17. *Let $\overset{tscd}{\rightsquigarrow} := \overset{\mathcal{P}_\top}{\rightsquigarrow}$ be the **termination-sensitive control dependence**.*

Proposition 10. $\overset{cd}{\rightsquigarrow} \subseteq \overset{tscd}{\rightsquigarrow} \subseteq \overset{wcd}{\rightsquigarrow}$.

Proof: Since $\mathcal{P}_\bot \subseteq \mathcal{P}_\top \subseteq \mathcal{P}_{\bot\infty}$, by Proposition 9 and Definition 17, $\overset{cd}{\rightsquigarrow} \subseteq \overset{tscd}{\rightsquigarrow} \subseteq \overset{wcd}{\rightsquigarrow}$. □

By Proposition 9, the set V_∞ acts as a "knob" tuning the precision of the control dependence relation. For example, if $V_\infty = \emptyset$ then the termination-sensitive control dependence becomes precisely the classic control dependence. If $V_\infty = V$ then it becomes the weak control dependence. In practice, V_∞ is somewhere in-between $\emptyset$ and V. However, the more nodes are added to V_∞, the more dependences are added, i.e., the weaker the dependence relation becomes. For example, in Figure 8 (C), suppose that $C_2 \notin V_\infty$. Then S_2 is not termination-sensitive control dependent on C_2. But if the user declares that $C_2 \in V_\infty$ despite of the actual semantics of the program, we will have $C_2 \overset{tscd}{\rightsquigarrow} S_2$.

Ideally, one would like to pick a V_∞ which would generate a *minimal* set of complete paths $\mathcal{P}_\top$ that includes all the actual execution paths of the program to analyze. Unfortunately, the selection of such an optimal V_∞ is difficult to achieve, because one would need to automatically prove termination of loops, an undecidable problem. A safe approach would be to start with $V_\infty = V$, and then remove from it all the statements which are not loop conditions, then all those loop conditions controlling terminating loops which can be detected by heuristic criteria or declared so by users or code generators.

Interestingly, there are no inclusion relations between the direct versions of these control dependences, that is, between $\overset{d\mathcal{P}_\bot}{\rightsquigarrow}$ (or $\overset{dcd}{\rightsquigarrow}$) and $\overset{d\mathcal{P}_\top}{\rightsquigarrow}$ or between $\overset{d\mathcal{P}_\top}{\rightsquigarrow}$ and $\overset{d\mathcal{P}_{\bot\infty}}{\rightsquigarrow}$ (or $\overset{dwcd}{\rightsquigarrow}$). For example, consider the CFG in Figure 8 (D). Suppose first that $C_2 \in V_\infty$ (i.e., the loop containing S_1 and C_2 is annotated as "non-terminating"). Then $C_1 \overset{d\mathcal{P}_\bot}{\rightsquigarrow} S_3$ but S_3 is not directly $\mathcal{P}_\top$-control dependent on C_1, while $C_2 \overset{d\mathcal{P}_\top}{\rightsquigarrow} S_2$ but S_2 is not directly control dependent on C_2. Suppose next that $C_2 \notin V_\infty$ (i.e., the loop containing S_1 and C_2 is not annotated as "non-terminating"). Then $C_1 \overset{d\mathcal{P}_\top}{\rightsquigarrow} S_3$ but S_3 is not directly weak control dependent on C_1, while $C_2 \overset{d\mathcal{P}_{\bot\infty}}{\rightsquigarrow} S_2$ but S_2 is not directly $\mathcal{P}_\top$-control dependent on C_2.

3 Control Scope

The *control scope* of a conditional statement is the set of statements that control depend on it, where the control dependence relation is *parametric* (in the sense of the previous section) and *indirect*. In other words, a statement S is the control scope of C iff the execution of S depends upon a fortunate choice made by C. Algorithms to compute the direct control dependence [13] and the direct weak control dependence [3] are well-known. These algorithms take linear time to detect all the statements that *directly* depend upon a given statement C, and can be used to construct program dependence graphs (PDG) [15], which are widely adopted in program slicing. These linear algorithms to calculate control dependencies are sufficient in applications where high online speed is not crucial and where only the direct dependencies are necessary, such as debugging. However, there are applications that need the transitive versions of the control dependences. For example, in [22], the (indirect) weak control dependence is used to prove total correctness of programs. Also, in predictive runtime analysis, one prefers to calculate all the dependencies statically and then spend constant time at runtime to check whether the statements generating two events depend upon each other, to reduce the runtime overhead.

Calculating all the direct dependencies for all the statements statically can therefore be achieved in $O(|V|^2)$. This also works for the termination-sensitive control dependence introduced in the previous section, because its parameter instance fits the framework in [3]. However, it is not clear how to effectively calculate the *indirect* control dependencies. A blind application of the transitive closure of the direct control dependence would yield an $O(|V|^3)$ algorithm (since the direct $\mathcal{P}$-control dependence is not a partial order), which can be inapplicable even on relatively small programs. Without any additional information about the program which generates the CFG, it seems that there is nothing that one can do to decrease the complexity of calculating the $\mathcal{P}$-control dependence. However, CFGs are typically generated from actual code that is stored as lines of sequences of characters in files. In what follows, we augment the CFG with code references and show that, under some rather common restrictions, we can calculate the entire $\mathcal{P}$-control dependence relation in $O(|V|^2)$, which is the same as the complexity of calculating the direct $\mathcal{P}$-control dependence. It may seem that $O(|V|^2)$ is still impractical in large applications; however, in the case of predictive runtime analysis or unit testing, we only need to calculate the control scopes for relatively small units, e.g., only intra-procedurely.

The nodes of a CFG generally correspond to either *simple statements* (i.e., statements that do not contain sub-statements) or to conditions that are parts of *compound statements* (i.e., statements that contain sub-statements). We only consider two types of compound statements in the sequel, namely conditionals and loops; note that although a programming language may also support other kinds of compound statements, e.g., `try..catch`, such statements are decomposed into simple statements when constructing the CFG. So they need not appear explicitly in the CFG (they appear only implicitly, encoded by correspoding edges). Even though CFGs capture faithfully the control flow of a program, unfortunately, precious structural information about the program, such as where a compound statement starts and where it ends, is generally not reflected in a CFG. In what follows we augument CFGs with structural information by adding to each node a corresponding unique line, or code reference number, which can be thought of as the position in the program where the statement corresponding to that node is located. The reference numbers of all nodes are assumed distinct. Since there is a one-to-one correpondence between (simple and compound) statements in the program and nodes in the CFG, we can identify statements with the reference numbers of their corresponding nodes in the CFG. Since the corresponding node in the CFG of a loop is its condition, the reference number corresponding to a statement is not necessarily the line number where that statement starts! For example, the reference number of the `do..while` loop in Fig. 10 (B) is 3. Let us formalize this:

Definition 18. *A **sequential CFG** $\langle V, E, START, END, \#, \flat \rangle$, abbreviated **SCFG**, is a CFG together with injective functions $\# : V \to \mathbb{N}$ and $\flat : V_C \to Intervals(\mathbb{N})$, such that*

1. *$\#(C) \in \flat(C)$ for any $C \in V_C$ and*
2. *$\flat(C') \subset \flat(C)$ for any $C \neq C' \in V_C$ with $\#(C') \in \flat(C)$.*

The function # associates to each node in the CFG, corresponding to either a simple statement (whose out-degree is 1) or a condition (whose out-degree is 2) that is part of a compound (i.e., conditional or loop) statement, a unique reference number. The function $\flat$ returns for each condition the code reference boundaries of its corresponding compound statement, given as an interval bounded by the smallest and the largest code reference numbers of nodes in the SCFG covered by that statement; some statements may include but not overlap other statements.

Fig. 10 shows the SCFGs for the programs in Fig. 9; we drew the nodes in ascending order of line numbers, and augumented every node with its reference number and each condition with its statement boundaries. The computation

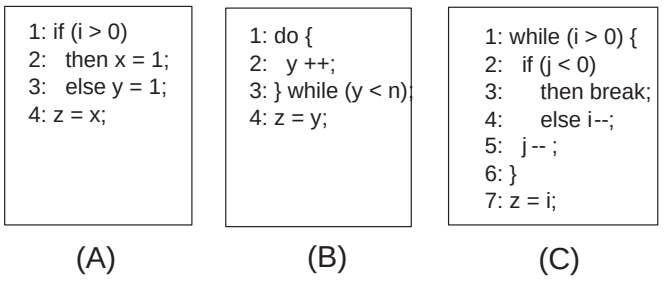

Fig. 9. Example Programs

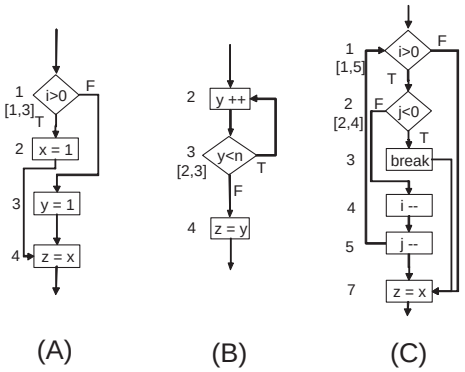

Fig. 10. Sequential CFGs of Fig. 9

of the $\flat$ function is straightforward and can be done at parse time at no additional complexity. For example, in Fig. 9 and 10 (A), $\flat(1) = [1, 3]$; in Fig. 9 and 10 (B), $\flat(3) = [2, 3]$; and in Fig. 9 and 10 (C), $\flat(1) = [1, 6]$. For each SCFG, one can define a function $next : V - V_C - \{END\} \to \mathbb{N}$, which associates to each node $S \in V - V_C - \{END\}$ the number $\#(S')$ where $(S, S') \in E$ is the unique outgoing edge from S. For "jump" statements, including `break`, `continue`, `return`, and exception throwing, *next* is the reference number of the statement that S jumps to; e.g., in Fig. 10 (C), $next(3) = 7$. If S is a simple non-jump statement at the end of a loop body, then $next(S)$ is the reference number of the loop statement; e.g., in Fig. 10 (B), $next(2) = 3$, and in Fig. 10 (C), $next(5) = 1$. For all other simple statements, the *next* function simply returns the reference number of the next statement in the program; e.g., in Fig. 10 (A), $next(2) = next(3) = 4$, and in Fig. 10 (C), $next(4) = 5$. Therefore, a simple non-jump statement is regarded as a jump to the next statement in the program.

As mentioned, we can identify statements in the program with their corresponding nodes in the SCFG. For this reason, from here on we take the liberty to call *all* the nodes in an SCFG statements and define the following natural terminology, based exclusively on the SCFG (with no reference to the code of the program):

Definition 19. *Nodes in V_C are called **compound statements** and those in $V - V_C$ are called **simple statements**. If C is compound and S any statement with $\#(S) \in \flat(C)$ then S is a **sub-statement** of C, or C **contains** S; if additionally there is no proper sub-statement C' of C that properly contains S then S is a **direct sub-statement** of C.*

The requirements of SCFGs are common to all programing languages that we are aware of. Most higher level structured programming languages, such as Java and C#, impose additional restrictions on jump statements; for example, `continue`, `break`, `return`, exception throwing, can only jump to specific positions determined statically at compile time. We next define a corresponding version of SCFG that captures formally the restrictions on jumps encountered in high-level structured programming languages:

Definition 20. *A **structured SCFG**, abbreviated as **SSCFG**, is an SCFG $\langle V, E, START, END, \#, \flat\rangle$ that satisfies:*

1. *Each compound statement has a unique entry point which is the lower bound of $\flat(C)$, written $entry(C)$; if $\#(S) \notin \flat(C)$ and $next(S) \in \flat(C)$ then $next(S) = entry(C)$;*
2. *Backward control flows can only be caused by loops: for any edge $(S, S') \in E$ with $\#(S) > \#(S')$, there exists a compount statment C such that $\#(S) \in \flat(C)$ and $\#(S') = entry(C)$; in this case we call C a **loop statement**. All compound statements which are not loop statements are called **conditional statements**. For every loop statement L, we also have a **next** function that points to the statement following L. Formally, $next(L) = max(\#(S_1), \#(S_2))$ where $(L, S_1)(L, S_2) \in E$.*

All the SCFGs in Fig. 10 are SSCFG; nodes with references 3 in Fig. 10 (B) and 1 in Fig. 10 (C) are loop statements. Note that even though one could technically define loop statements in the context of (unstructured) SCFGs, they make full sense only in the context of SSCFG, because in a SCFG one can construct a "loop statement" using a branch statement (e.g., a `if` statement) with an arbitrary jump (`goto`) statement.

We next focus on computing the control scope function of compound statements. Ideally, the control scope of a compound statement C would contain precisely the statements that are control-dependent on C. Unfortunately, such

statements can be spread all over the program, thus making their precise bookkeeping rather chanllenging. However, in what follows we show that in the context of SSCFG, the statements that are control dependent on a compound statement C are all located into a *window*, or *interval*, of references, say *scope* (C), with the property that *scope* (C) contains no (reference of) statements that are *forward-reachable* (see Definition 22) from but not control dependent on C. In other words, the interval, *scope* (C), characterizes unambiguously all the statements that are control-dependent on C; if one wants to find precisely those statements control-dependent on C, all one needs to do is to perform a simple (linear) reachability analysis from C and then all the statements in *scope* (C) that are not control-dependent on C can be easily filtered out. Moreover, in many applications one will only be interested in checking dependency for statements S that are, for external reasons, known to be reachable from C. For example, in our runtime analysis approach, we need to check for control dependence of the statements that generated an *observed* event e_s upon the compound statement C that previously generated an event e_c; therefore, the very existance of the event e_s as after the event e_c is a proof of reachability of S from C. For such applications, our technique below to calculate control scopes can be very effective, because the checking control dependence reduces to checking membership to an interval. Moreover, we show, that one can calculate all the control scopes of a SSCFG in $O(|V|^2)$, instead of $O(|V|^3)$ that is needed for an unrestricted CFG.

We can easily see that all sub-statements of a compound statement are control dependent on it. Besides, a jump statement from within a compound statement C may extend the control scope of C. For example, in Fig. 10 (C), the `break` statement extends the scope of the `if` statement to the end of the loop, therefore statement 5 is control-dependent on the compound statement 2. This can be formalized as follows:

Definition 21. *Suppose that C is a compound statement with $\flat(C) = [b_1, b_2]$. Then we define the **pre-scope** of C, written pre-scope(C), as follows:*

1. *If C is a conditional statement then pre-scope(C) is $[b_1, max(b_2, next(J_1)-1, .., next(J_n))-1]$, where J_i for $i \in [1,n]$ are all the direct sub-statements of C; and*
2. *If C is a loop statement, pre-scope(C) = $\flat(C)$.*

For example, in Fig. 10 (C), the pre-scope of the loop is $[1,6]$ while the pre-scope of the `if` statement is $[2,6]$. The pre-scopes of loop statements are not extended by their direct sub-statements (when, e.g., an exception is thrown or a break/continue for an outer loop) because, as we discuss below, the backward edges of loops cause a different situation to handle. The pre-scopes of statements can be easily calculated by at no additional cost at parse time because the targets of jump statements are known statically (we focus on intra-procedure analysis here; if a statement throws an exception that is not caught in the analyzed procedure, it is assumed to jump to the end of the procedure). Note that the pre-scope of C may already contain statements that are *not* control-dependent on C. Considering the example in Fig. 11: the pre-scope of the conditional statement 3 is $[3,8]$ because of the `continue` statement in one of its branches. So statement 8 is within its pre-scope, but obviously not control-dependent on it. To filter out such statements, we next introduce a new relation between statements:

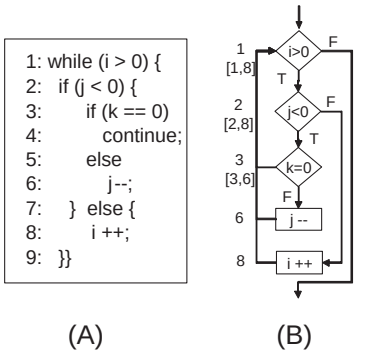

Fig. 11. Example about extended pre-scope

Definition 22. *Statement S' is **forward-reachable** from S iff there exists an $S - S'$ path that contains no loop statement containing both S and S'.*

For example, in Fig. 10 (C), node 3 is reachable but not forward-reachable from 4, and in Fig. 11, statement 8 is reachable but not forward-reachable from statement 3. Although the intuition for forward-reachability is that from S "one can go forward and reach" S', it is *not* always the case that one can find an $S - S'$ path with increasing reference numbers. For example, in Fig. 11, statement 8 is forward-reachable from statement 2, but the path between them always contains statement 1, in other words, there is no path from 2 to 8 with increasing reference numbers. Forward-reachability can be determined by the following proposition:

Proposition 11. *Given an SSCFG G and statements S and S', S' is forward-reachable from S iff S' is reachable form S in a graph G' contructed by transforming G as follows: for every (back) edge $e = (n_1, n_2)$ in G with $n_1 > n_2$, corresponding to loop L (that is, entry(L) = n_2), replace e by $(n_1, @-(L))$.*

Proof: First, it is obvious that all paths in G' contain only increasing reference numbers and G and if S' is reachable from S in G' then S' is reachable from S in G. Suppose that S' is not forward-reachable from S in G. If S' is not reachable from S in G then S' is not reachable from S in G' either. If there exist $S - S'$ paths then all of them contain some loop L which contains both S and S'. In other words, all $S - S'$ paths contain an edge $e = (n_1, \#(L)), n_1 > \#(L)$. This edge is replaced with (n_1, n_3) in G' where $n_3 > \#(S')$, which means that S' is not reachable from statement n_3 in G'. So one cannot find a $S - S'$ in G', that is to say, S' is not reachable from S in G'.

Now suppose that S' is foward-reachable from S in G and π is a $S - S'$ path that contains no loop that contains both S and S'. Then for every loop L contained in π, we keep only one iteration of L; if L contains S or S' then the iteration to keep should go trhough S or S' correspondingly. If the loop exits at its entry, i.e., the `while` loop, then the path contains a sequence of edges $(n_1, \#(L)), (\#(L), n_2)$ where $n_1 > \#(L))$ and n_2 is the reference number of the statement following L. We then replace these two edge by (n_1, n_2) in G'. This way, we construct a $S - S'$ path in G', that is to say, S' is reachable from S in G'. □

Proposition 11 gives a very simple and effective to compute forward-reachability. Now we can have the following property for the pre-scope:

Proposition 12. *For a (simple or compound) statement S and a compound statement C, if $\#(S) \in$ pre-scope(C) and S is forward-reachable from C, then $C \rightsquigarrow S$.*

Proof: If C is a loop statement, since the pre-scope of C is $\flat(C)$, any statement in the pre-scope is control-dependent on C. Suppose that C is a conditional statement and S falls in the pre-scope of C and is forward-reachable from C. If $\#(S) \in \flat(C)$, then $C \rightsquigarrow S$. If S is out of $\flat(C)$ (so the pre-scope of C is larger than $\flat(C)$) and S is not control-dependent on C then all $C - S$ paths contain $ipd_{\mathcal{P}}(C)$. Obviously, $ipd_{\mathcal{P}}(C)$ is outside of $\flat(C)$. Let b be the upper bound of $\flat(C)$, then, by Definition 21, there exists a statement S' such that $\#(S') = b$ and we can find a $C - S'$ path that does not contain any node within the pre-scope of C but out of $\flat(C)$. If $\#(ipd_{\mathcal{P}}(C)) < b$, then there exists an $S' - ipd_{\mathcal{P}}(C)$ path π which contains a loop L that contains both S' and $ipd_{\mathcal{P}}(C)$. Then $ipd_{\mathcal{P}}(C)$ must be the node corresponding the loop; otherwise, the loop can choose to exit and skip $ipd_{\mathcal{P}}(C)$ which is impossible. Moreover, L should contain C; otherwise, there exists a jump from outside of $\flat(L)$ into $\flat(L)$, contradicting to our assumptions on SSCFG. So every $C - S$ path contains L, contradicting to the hypothesis. If $\#(ipd_{\mathcal{P}}(C)) \geq b$ then any $ipd_{\mathcal{P}}(C) - S$ path contains a loop L that contains $ipd_{\mathcal{P}}(C)$ and S. Similarly, L contains C because of our assumptions on SSCFG. So any $C - S$ path contains L, contradiction. □

Definition 23. *For a compound statement C, a control scope of C is a an interval of reference numbers with the following properties:*

1. *all statements that are control dependent on C are contained in the interval;*
2. *if a statements S is within the interval and not control dependent on C then S is not forward-reachable from C.*

For every compund statement, there can be multiple control scopes. In what follows we show that such control scopes exist and give an $O(|V|^2)$ algorithm to compute one of them. And by control scope, we mean the one computed by our algorithm from now on. The control scope of a compound statement can be larger than its pre-scope because of pre-scopes may overlap. Considering the control scope of the `if` statement in Fig. 10 (C), its control scope is $[1, 6]$ because its pre-scope overlaps the one of the outer loop.

To facilitate the following discussion, we abstract the sequential CFG to emphasize the pre-scopes of statements, as shown in Fig. 12. In this figure, the ranges of arrows give the pre-scopes of the statements, while the directions of the arrows distinguish the branch statement and the loop statement, that is, the forward arrow represents the branch statement and the backward one for the loop statement. According to the assumptions that we have, there are only two cases for overlapped pre-scopes, as shown in Fig. 12 (A) and (B). In the first case, C_2 is forward reachable from C_1. Then the control scope of C_1 is extended by that of C_2, because of the transitivity of control dependence. Consider the statement S_1 that resides out of the pre-scope of C_1. C_1 may choose to go into the branch containing C_2 and then S_1 will be skipped, that is, $C_1 \overset{\mathcal{P}}{\rightsquigarrow} S_1$. In the second case C_1 and C_2 have the same control scope because of the backward jump caused

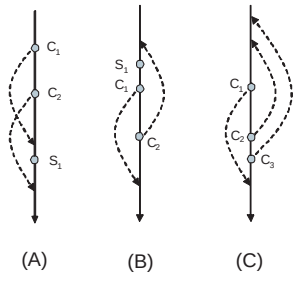

Fig. 12. Overlapped Pre-scopes

by the loop C_2. For example, consider the statement S_1 before C_1. Its execution in the second iteration of the loop is dependent on the choice made at C_1 in the first iteration.

Based on the above observation, Fig. 13 shows the algorithm to compute the $scope_{\mathcal{P}}$ function in $O(|V|^2)$ time based on the above observations. The computation process consists three steps. The first step is to extend pre-scopes of statements as in Fig. 12 (A) by a backward scanning. The second step is to compute equivalence classes of statements that have the same control scope by checking overlapped forward and backward conditionals (Fig. 12 (B)). We first build a graph to represent the overlapping among loops and other conditionals and then calculate connected components in the graph, which are essentially the desired equivalence classes. At the end, we compute the scopes of obtained equivalence classes. Note that this step can be adapted to compute different $\mathcal{P}$-control scopes. For example, for the classical control dependence, the `endln` will never be set to `infinity`; while for the weak control dependence, the `endln` will be always `infinity` whenever the equivalence class contains a loop.

```
procedure ComputeScope()
  ComputeFWReachability();
  ExtendPreScope();
  BuildEquivalentClasses();
  ComputeEquivalentClassScope();
endProcedure
procedure ComputeFWReachability()
  transform the original CFG into the corresponding non-loop CFG;
  for every statement S in the program do
    use depth-first search to compute the set of forward-reachable statements of S
        and the set of statements which can forward-reach S;
  endFor
endProcedure
procedure ExtendPreScope()
  for S = the last statement downto the first statement do
    if (S is a non-loop conditional) then
      for every non-loop conditional S' that can forward-reach S do
        if prescope(S) overlaps prescope(S') then
          prescope(S') = prescope(S') U prescope(S);
        endIf
      endFor
    endIf
  endFor
endProcedure
procedure ComputeEquivalentClasses()
  create a graph G containing nodes correpsponding to conditionals;
  for every loop L do
    for every non-loop conditional C in prescope(L) do
      if (prescope(L) overlaps prescope(C)) then
        create an edge between L and C in G;
      endIf
    endFor
  endFor
  compute connected component in G;
  for every connected component Cls do
    for every statement S in Cls do
      set(S) = Cls;
    endFor
  endFor
endProcedure
procedure ComputeEquivalentClassScope();
  for every connected component Cls do
    beginln = the smallest lower bound of pre-scopes of statements in Cls;
    if Cls contains at least one non-terminating loop then
      endln = infinity; //infinity is the maximum integer in the system
    else
      endln = the largest upper bound of pre-scopes of statements in Cls;
    endIf
    scope(Cls) = [beginline, endline];
  endFor
endProcedure
```

Fig. 13. Compute the scope function

The output of this algorithm includes a $prescope_{\mathcal{P}}$ function that maps an equivalence class into its control scope, and a function *set* that maps a statement into the corresponding equivalence class. One can show that:

Lemma 8. *For a conditinal statement C and a statement S, if S is outside of scope* ($sets[C]$)*, then S is not $\mathcal{P}$-control dependent on C.*

Proof: If S is not reachable from C then S is not control-dependent on C. Suppose that S is reachable from C, then there exists a $C - S$ path π. If S is before C then π contains a loop L containing both C and S. Since S is outside of

scope (C), L is outside of *scope* (C) too. Then L is a post-dominator of C. So S is not control-dependent on C. If S is after C and control dependent on C then by the definition of control dependence, there exists a $C - END$ path π' that does not contain S. π' must contain at least an edge (S_1, S_2) with $\#(S_1) \in scope(C)$ and $\#(S_2) > \#(S)$. Let e be the first such edge in π'. If S_1 in e is forward-reachable from C or within a loop that has the same scope with C then by the algorithm, $\#(S) \in scope(sets[C])$. So $\#(S) \in scope(sets[C])$, contradiction. So any $C - S_1$ path π'' should contain a loop L containing both C and L is outside of *scope* ($sets[C]$), which means that π'' contains an edge that jumps from inside of *scope* ($sets[C]$) to the end of L. If S is in L then by the algorithm, $\#(S) \in scope(sets[C])$, contradiction; but if S is outside of L then any $C - S$ path should contain L, so S is not control dependent on C. □

Proposition 13. *Under the assumptions of statements above, for a conditional statement C and a statement S, $C \rightsquigarrow S$ iff $\#(S) \in scope(sets[C])$ and one of the following holds:*

1. *S is forward-reachable from C or*
2. *there exists a loop L such that $sets[C] = sets[L]$ and $S \in \flat(L)$.*

Proof: If S is control dependent on C then $\#(S) \in scope(sets[C])$ by Lemma 8. If there exists no loop L such that $sets[C] = sets[L]$ and $S \in \flat(L)$ and S is not forward-reachable from C. Then any $C - S$ path π contains a loop L' containg C and S and L' is outside of *scope* ($sets[C]$). Then L' is a post-dominator of C, so S is not control dependent on C, contradiction.

Suppose that $\#(S) \in scope(sets[C])$. If there exists a loop L such that $sets[C] = sets[L]$ and $S \in \flat(L)$ then S is obviously control dependent on C. Otherwise, if S is forward-reachable from C and not control dependent on C then any $C - S$ path contains $ipd(C)$. So $\#(ipd(C)) \in scope(sets[C])$, which is impossible according to the algorithm. □

The complexity of *ComputeFWReachability()*, *ComputePreScope()* and *ComputeEquivalentClasses()* is $O(|V|^2)$ and *ComputeEquivalentClassScope()* is $O(|V|)$. So the overall complexity of this algorithm is $O(|V|^2)$.

4 Sliced Causality

Based on the control scope function, we are able to define the hybrid dependence on events, which are then used to slice the communication among threads. This way, one can achieve a more relaxed causal partial order on events, which we call sliced causality.

4.1 Events and Traces

Events play a crucial role in our approach, representing atomic steps in the execution of the program. An event can be a write/read on a location, the beginning/ending of a function invocation, acquiring a lock, etc. A statement in the program usually produces multiple events. Events need to store enough information about the program state in order for the observer to perform its analysis. Therefore, we define the notion of events in our approach as follows:

Definition 1 *An **event** is a mapping of **attributes** into corresponding **values**. Let Events be the set of all events. A **trace** is a finite sequence of events. We assume an arbitrary but fixed trace τ, let ξ denote the set of events in τ (also called **concrete events**), and let $<_\tau$ be the total order on ξ: $e <_\tau e'$ iff e occurs before e' in τ.*

For example, one event can be $e_1 : (id = 17897, thread = t_1, stmt = L_{11}, type = write, target = a, state = 1)$, which is a write on location a with value 1, produced at statement L_{11} by thread t_1. One can easily include more information into an event by adding new attribute-value pairs. We use *attribute*(e) to refer to the value of *attribute* of event e. To distinguish events with identical attributes, events are assigned unique identifiers when generated.

When the trace τ is checked against a property φ, most likely not all the attributes of the events in ξ are needed; some events may not even be needed at all. For example, to check data races on a variable x, the *states*, i.e., the values of x, of the events of type *write* and *read* on x are not important; also, updates of other variables or function call events are not needed at all. We next assume a generic *filtering function* that can be instantiated, usually automatically, to concrete filters depending upon the property φ under consideration:

Definition 2 *Let α_φ: $\xi \rightarrow$ Events be a partial function, called a **filtering function**. The image of α_φ, that is $\alpha_\varphi(\xi)$, is written more compactly ξ_φ; its elements are called **abstract relevant events**, or simply just **relevant events**.*