RC 21493 (96967) 28MAY99 Computer Science/Mathematics

Research Report

Context-Sensitive Synchronization-Sensitive Analysis is Undecidable

G. Ramalingam
IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598

IBM Research Division
Almaden - Austin - Beijing - Haifa - T. J. Watson - Tokyo - Zurich

1 Introduction

This paper illustrates one source of difficulty in statically analyzing concurrent programs. In the realm of static interprocedural analysis, one talks of context-sensitive analyses [14]: these are analyses that are precise with respect to calling-context, in a formally defined way. In a concurrent program that contains synchronization constructs, one can similarly define the notion of synchronization-sensitive analyses [15]: these are analyses that are precise with respect to the synchronization structure of the program. Efficient context-sensitive analysis algorithms have been developed for a number of analysis problems (*e.g.*, see [11, 13]). Taylor [15] shows that intraprocedural synchronization-sensitive analysis of concurrent programs is NP-hard. Consequently, intraprocedural synchronization-sensitive analyses can be expensive, but it is straight forward to show that they are possible for a number of analysis problems. In this paper we address the problem of interprocedural analysis of concurrent programs. Specifically, we show that context-sensitive *and* synchronization-sensitive analysis is impossible even for the simplest of analysis problems.

2 The Problem

In this section we present a simple description of the problem we are addressing. A more formal definition of the problem appears in the Appendix.

We are interested in the static analysis of concurrent programs. We use the programming model described by Taylor [15], and also used by Naumovich and Avrunin [8]. The model is a simple one, where a concurrent program consists of a set of tasks, which is statically specified. (Thus, new tasks can not be dynamically created during program execution.) A task is essentially like a sequential (non-concurrent) program, consisting of a collection of (possibly recursive) procedures, with a distinguished *main* procedure. (Taylor [15] focuses on the special case where each task consists of a single non-recursive procedure.) The only concurrency primitive we consider is that of (rendezvous style) synchronization: a task T may *invoke* an *entry* $S.E$ belonging to a task S and a task T may *accept* an entry belonging to T. Note that the entry invoked or accepted is statically specified. (Thus, entries are *not* first-class values.) This model corresponds to a simple subset of Ada. (Most concurrent progamming languages provide for several more powerful concurrency primitives. The undecidability result established in this paper applies to any language that includes at least the above features, or other equivalent features.)

The semantics of the synchronization primitives are as follows. A task which invokes an entry $S.E$ is suspended until the task S *accepts* the entry E, after which both tasks proceed with their execution. If the task S performs an *accept* of entry E when no other task has invoked the entry $S.E$, it is suspended until some other task invokes the entry E, after which both tasks are free to proceed with their execution.

An execution of a concurrent program consists of the execution of each of its tasks (*i.e.*, the *main* procedure of each of its tasks). The execution of each task proceeds sequentially. Except for the constraints imposed by the semantics of the synchronization primitives, statements belonging to different tasks may be executed in any order.

The goal of various static analyses is to compute information about the possible behavior of the program at runtime. The nondeterminism inherent in concurrent programs can complicate static analysis. If the order in which statements belonging to different tasks are executed can affect the program behavior and the values computed, then static analysis has to conservatively account for all possible orders in which statements of different tasks might execute. Since the synchronization primitives impose some constraints on the order in which different statements may execute, static analyses that take these constraints into account can compute more precise results than those don't.

The following example (in Ada syntax) illustrates this:

```
task body X is begin
    [1] V := 0;
    [2] invoke Y.E;
    [3] invoke Y.E;
    [4] Z := V;
end X;

task body Y is begin
    [5] accept E;
    [6] V := 1;
    [7] accept E;
end Y;
```

In an execution of the above example program, statements [2] and [5] will synchronize with each other. Similarly, the execution of statements [3] and [7] will be synchronized. Consequently, the execution of statement [1] is constrained to occur before the execution of statement [6], which is constrained to occur before the execution of statement [4]. Hence, the value of variable `V` in statement [4] will be the value `1` assigned in statement [6]. Static analysis that takes the synchronization constraints into account can infer this. An analysis of this program that ignores the synchronization constraints, however, will have to assume that the statements [1] and [6] may execute in any order, and, hence, yield the less precise information that the value of variable `V` in statement [4] might be either `0` or `1`.

We say that a static analysis is synchronization-sensitive if it takes the constraints imposed by the synchronization statements in the program into account. A more formal definition of this concept appears in the Appendix. The (partial) execution of a concurrent program can be represented by a collection of (partial) execution paths, one for each task in the program. We refer to such a collection of execution paths as a *potential execution sequence* for the program. An execution path in a task contains a sequence of *synchronization events*, where a synchronization event is either the invocation or acceptance of an entry. Not every potential execution sequence can may occur in practice. In particular, the semantics of synchronization rules out some potential execution sequences as being infeasible. We say that a given potential execution sequence is a legal execution sequence if the synchronization events in the sequence can be paired up in a consistent manner. (We refer the reader to the Appendix for a more detailed and formal definition of these concepts.) A static analysis is said to be synchronization-sensitive if it treats potential execution sequences that are not legal as unexecutable and computes information describing only execution along legal execution sequences (similar to the meet-over-all-paths analyses [5] and meet-over-all-valid-paths analyses [14].)

Synchronization-sensitive analysis is not easy. The synchronization constraints can impose not only an order on the execution of statements belonging to different tasks, it can also rule out certain paths within a single task as being unexecutable The following example illustrates this:

```
task body X is begin
    [1] invoke Y.E;
    [2] if (···) then
    [3]     invoke Y.E;
    [4]     W := 0;
    [5] else
    [6]     W := 1;
    [7] end if;
    [8] Z := W;
end X;

task body Y is begin
    [9] accept E;
end Y;
```

In this example, the invocation of entry `E` in statement [1] is accepted by statement [9]. If task `X` takes the true branch of the conditional in line [2], then the execution of [3] will suspend task `X` forever since the entry invocation will not be accepted by task `Y`. Hence, if execution reaches statement [8] then the false branch *must* have been taken in line [2]. Hence, a *synchronization sensitive* analysis of the above program can establish that the value of `W` in line [8] must be `1`. The above example illustrates the "power" of synchronization sensitive analysis and it should come as no surprise that such analyses can be expensive. Taylor [15] shows that various such analyses are NP-complete in the intraprocedural case, where each task consists of a single non-recursive procedure.

Though potentially expensive, such analyses are possible in the intraprocedural case. Assume that each task is represented by a control-flow graph. Note that for any such multi-tasking program we can create a "product graph" of the control-flow graphs of the tasks in the program. A vertex in the product graph corresponds to a tuple consisting of a vertex in every task's graph. We have some choices in terms of how the set of edges of the product graph is defined. Consider the product graph that contains an edge from a tuple $(u_1, \cdots, u_k)$ to a tuple $(v_1, \cdots, v_k)$ iff every v_i is either equal to u_i or is a successor of u_i and at least one v_i is a successor of u_i. Paths in this product graph represent essentially the set of all potential execution sequences through the given program. On the other hand, it is straight forward to define the set of edges in the product graph such that paths in the product graph represent essentially the set of all legal or synchronization sensitive execution sequences through the given program. We only need to eliminate those edges that cause a synchronizing vertex u_i to transition to its successor vertex v_i without a matching synchronizing vertex making its transition. We can use this product graph to directly adapt intraprocedural analyses of sequential programs to analyze concurrent programs in a synchronization sensitive way.

The focus of this paper is on the interprocedural analysis of concurrent programs, where each task consists of a set of (possibly recursive) procedures

The question we are interested in is how *procedures* complicate such synchronization-sensitive analysis. In particular, assume that each task consists of a set of (possibly recursive) procedures, with a distinguished *main* procedure. Is it feasible to perform *interprocedural* static analyses of concurrent programs in a *context-sensitive and synchronization-sensitive* way? (An interprocedural analysis is said to be context-sensitive if it considers only paths with no mismatching call and return edges. A more complete definition appears in the Appendix. We say that a potential execution sequence is context-sensitive if every path in the sequence is context-sensitive.)

One of the simplest static analysis problem is to determine if program execution might ever reach a particular vertex. We define this *reachability* problem as follows.

Definition 1 (The context-sensitive and synchronization-sensitive reachability problem) *Given a program P consisting of a set T of tasks, a task τ in T, and a program point u in τ, determine if there exists a context-sensitive legal execution sequence $(\pi_{\tau'} : \tau' \in T)$ for P, where every path $\pi_{\tau'}$ starts at the start vertex of the main procedure of τ' and the path π_τ ends at program point u.*

Taylor defines several other analysis problems for concurrent programs. The goal of the "Possible Rendezvous" (or PR) problem is to determine if two given statements in a concurrent program may possibly rendezvous (*i.e.* synchronize with each other). Another problem closely related to the PR problem is the "May Happen in Parallel" (or MHP) problem, which is to determine if two given statements in a concurrent program may ever execute in parallel. We can, in an analogous fashion, also define generalizations of various analysis problems that have been studied in the context of sequential programs, such as the "reaching definitions" problem.

The context-sensitive and synchronization-sensitive reachability problem defined above is simpler than the context-sensitive and synchronization-sensitive versions of most analysis problems, including the PR problem, the MHP problem, and the reaching definitions problem. In other words, the reachability problem can be trivially reduced to these other problems. Consequently, the undecidability of the reachability problem (established in the next section) implies the undecidability of these other analyses.

3 An Undecidability Result

In this section we show that identifying the existence of context-sensitive synchronization-sensitive paths is impossible, implying the imposibility of performing static analysis of concurrent programs in a context-sensitive and synchronization-sensitive way.

Definition 2 *The Post's Correspondence Problem or PCP is the following: Given arbitrary lists A and B of r strings each in $\{0,1\}^+$, say*

$$A = w_1, w_2, \cdots, w_r$$
$$B = z_1, z_2, \cdots, z_r$$

does there exist a nonempty sequence of integers i_1, i_2, $\cdots$, i_k such that

$$w_{i_1} w_{i_2} \cdots w_{i_k} = z_{i_1} z_{i_2} \cdots z_{i_k}$$

Theorem 1 *The Post's Correspondence Problem is undecidable. [4]*

Theorem 2 *The context-sensitive synchronization-sensitive reachability problem is undecidable.*

```
task D is
    entry allDone_A, allDone_B;
end D;

task A is
    entry W_1, W_2, ···, W_r;
end A;

task B is
    entry Z_1, Z_2, ···, Z_r;
    entry bit_0, bit_1;
end B;
```

```
task body A is
    procedure F is
        if (···) then
            return ;
        elsif (···) then
            accept W_1;
            F();
            // Encode string w_1 in reverse.
            // Note: w_1[|w_1|] denotes the last bit of w_1.
            invoke B.bit_{w_1[|w_1|]}; ···; invoke B.bit_{w_1[1]};
        elsif (···) then
        ···
        else
            accept W_r;
            F();
            invoke B.bit_{w_r[|w_r|]}; ···; invoke B.bit_{w_r[1]};
        end if;
    end F;
begin
    F();
    invoke D.allDone_A;
end A;
```

```
task body D is begin
    repeat
        if (···) then
            invoke A.W_1; invoke B.Z_1;
        elsif (···) then
            invoke A.W_2; invoke B.Z_2;
        ···
        else
            invoke A.W_r; invoke B.Z_r;
        end if;
    until (···);
    accept allDone_A;
    accept allDone_B;
    // The following statement is reachable
    // iff the given PCP instance has a solution
    u:
end D;
```

```
task body B is
    procedure H is
        if (···) then
            return ;
        elsif (···) then
            accept Z_1;
            H();
            // Encode string z_1 in reverse.
            // Note: z_1[|z_1|] denotes the last bit of z_1.
            accept bit_{z_1[|z_1|]}; ···; accept bit_{z_1[1]};
        elsif (···) then
        ···
        else
            accept Z_r;
            H();
            accept bit_{z_r[|z_r|]}; ···; accept bit_{z_r[1]};
        end if;
    end F;
begin
    H();
    invoke D.allDone_B;
end B;
```

Figure 1: An instance of the reachability problem generated from an instance of Post's Correspondence Problem. The exit vertex of task `D` is reachable along a context-sensitive and synchronization-sensitive execution sequence iff the given PCP instance has a solution.

Proof We will show a reduction from the Post's Correspondence Problem to the context-sensitive synchronization-sensitive reachability problem. In particular, given an instance

$$A = w_1, w_2, \cdots, w_r$$
$$B = z_1, z_2, \cdots, z_r$$

of PCP, consider the multi-tasking program (written in Ada) shown in Fig. 1.

This program consists of three tasks D, A, and B. We can establish a bijective correspondence between complete execution paths in task D and non-empty sequences of integers i_1, i_2, $\cdots$, i_k, where every i_j lies between 1 and r (inclusive). In particular, any execution path that iterates through the repeat-until loop k times corresponds to a sequence i_1, i_2, $\cdots$, i_k where i_j indicates which of the r branches inside the loop was taken during the j-th iteration of the loop. One can think of task D as "generating" all possible integer sequences (of the above form) and "communicating" the generated sequence to tasks A and B.

Task A consists of a recursive procedure F. Procedure F "receives" the sequence generated by task D, and utilizes the recursive call-chain to "remember" this sequence. Once a sequence i_1, i_2, $\cdots$, i_k is received, procedure F "generates" the string $w_{i_1}w_{i_2}\cdots w_{i_k}$ in reverse and "communicates" this generated string to task B.

Task B also consists of a recursive procedure H. Procedure H is very similar to procedure F. It "receives" the sequence generated by task D, and utilizes the recursive call-chain to "remember" this sequence. It then determines the string $z_{i_1}z_{i_2}\cdots z_{i_k}$ in reverse and is ready to "accept" this string from task A.

If $w_{i_1}w_{i_2}\cdots w_{i_k}$ is a suffix of $z_{i_1}z_{i_2}\cdots z_{i_k}$, then procedure F will terminate and task A will signal task D that it is all done by invoking entry D.allDone$_A$. Similarly, B will invoke entry D.allDone$_B$ if $z_{i_1}z_{i_2}\cdots z_{i_k}$ is a suffix of $w_{i_1}w_{i_2}\cdots w_{i_k}$. In particular, A will invoke D.allDone$_A$ and B will invoke D.allDone$_B$ iff $w_{i_1}w_{i_2}\cdots w_{i_k}$ is equal to $z_{i_1}z_{i_2}\cdots z_{i_k}$.

Thus, if the given PCP instance has a solution, then the exit vertex (the statement labelled u) of task D is reachable along a context-sensitive synchronization-sensitive execution sequence. Conversely, if the exit vertex of task D is reachable along a context-sensitive synchronization-sensitive execution sequence, then the given PCP instance has a solution.

It follows that the context-sensitive synchronization-sensitive reachability problem is undecidable. □

An Alternative Proof

We briefly mention an alternative reduction that can be used to establish the above undecidability result, as it offers some more insight into the difficulties of synchronization-sensitive analysis. It is well known that it is undecidable if the languages defined by two arbitrary context-free grammars have a non-empty intersection [4, Theorem 8.10]. Given a context-free grammar G_1, the grammar can be encoded as a process as follows: represent each terminal of the grammar by an entry, represent each non-terminal of the grammar by a procedure consisting of a multi-way conditional statement consisting of a branch for every production of that non-terminal. The branch encodes the right-hand side of the corresponding production, encoding the occurrence of a terminal by an invocation of the corresponding entry, and encoding the occurrence of a non-terminal by a call to the corresponding procedure. Execution of any (context-sensitive) path from the start vertex to the exit vertex of a procedure results in a sequence of entry invocations that "belongs" to the language defined by the non-terminal corresponding to the procedure. Conversely, for any string in the language defined by the non-terminal, there exists a (context-sensitive) path from the start vertex to the exit vertex of the procedure whose execution results in the corresponding sequence of entry invocations. Thus, the "entry invocations" performed by the procedures of the above process correspond to the languages defined by the non-terminals of G_1. Dually, one can encode a context-free grammar G_2 by a system of procedures that perform a sequence of "entry acceptances" corresponding to the languages defined by the non-terminals of G_2. Thus, it is straight-forward to define a program consisting of two processes that will terminate successfully iff the languages defined by the two given context-free grammars have a non-empty intersection.

4 Discussion

It is well known that "precise static program analysis" is impossible (*e.g.*. through connections to the halting problem). The result established in this paper, however, says more than that, and we briefly elaborate on this issue here.

All static program analysis algorithms combat undecidability by working with a *nonstandard* semantics of programs, which serves as an abstraction or approximation of the standard semantics of programs [5, 1]. Given a nonstandard semantics, one can talk of the most precise analysis with respect to that semantics, referred to as the collecting interpretation with respect to the nonstandard semantics, or the meet-over-all-paths solution [5]. (We are being somewhat loose with our terminology here for the sake of brevity.) One question that arises in designing static analysis algorithms is what kind of approximations one must make in the nonstandard semantics to guarantee that the most precise analysis with respect to that semantics is computable.

Consider procedure-less sequential programs. If the set of program states in the nonstandard semantics is finite (or, in Kildall's formalism, if the "lattice" is finite), then the most precise analysis with respect to the semantics is computable (using iterative techniques, for example).

In the presence of (potentially recursive) procedures, the program's state, in the standard semantics, consists of two components, one of which is the "call stack". We will refer to the second component of the program state as the "global state", for the lack of a better term. (Note that the "call stack" is only a sequence of procedures; if the procedures have local variables, we consider them part of the "global state".) One can construct nonstandard semantics that approximate the global state but not the call stack. The set of program states in such a nonstandard semantics is infinite since the call stack can grow unbounded. Interestingly, the most precise analysis with respect to such a nonstandard semantics is still computable as long as the set of global states in the nonstandard semantics is finite (as shown by Sharir and Pneuli [14]). Such analyses are precise with respect to the calling-context and are referred to as context-sensitive analyses.

Now consider concurrent programs without procedures. One can construct nonstandard semantics for such programs by approximating the state of each task, but preserving the synchronization semantics. The most precise analysis with respect to such a nonstandard semantics is computable if the set of abstract states for each task is finite. This leads to the notion of synchronization-sensitive analysis.

Unfortunately, these positive results fail to hold if we consider static analysis of concurrent programs with recursive procedures. The result in this paper shows that if a nonstandard semantics approximates neither the synchronization semantics of the program nor the call stack, then the most precise analysis with respect to that semantics is not computable.

In terms of related work, Reif [10] considers the problem of intraprocedural analysis of a system of communicating processes. In Reif's model, processes transmit and receive messages through asynchronous communication channels. Reif shows that reachability analysis is undecidable in the case of *dynamic* communication systems, where channels are first class values, and that reachability analysis is EXP-SPACE hard in the case of *static* communication, where the channels specified in "receive" and "transmit" statements are constants. Our result shows that the problem is undecidable even in the case of static communication in the inter-procedural case. The model we use is essentially an interprocedural extension of the model Taylor [15] uses. Taylor shows that several (intraprocedural) analysis problems are NP-complete in his model.

5 Conclusion

The precision of static analysis of concurrent programs, especially programs where different tasks (threads) may access and modify "shared" variables, can obviously be improved by taking into account the sequencing constraints imposed on the statements of the different tasks (threads) by the concurrency primitives. Several people (*e.g.*, see [2, 6, 3, 7, 8, 9]) have proposed various conservative algorithms for identifying the sequencing constraints on the statements of a concurrent program. This paper shows that identifying such sequencing constraints in a context-sensitive way is impossible. The proof technique used in this paper was inspired by Reps' result [12] which shows that context-sensitive, structure-transmitted data-dependence analysis is impossible. The actual undecidability result may not be very surprising once one understands the problem, but we believe that it is useful to understand these limitations when designing static analyses. We are not aware of any equivalent result in the literature, despite the importance of this problem and the amount of work that has gone into designing (conservative) static analyses for concurrent programs.

Acknowledgements

I would like to thank Bard Bloom for his helpful comments on the paper.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[2] E. Duesterwald and M. L. Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Proceedings of the ACM SIGSOFT Fourth Workshop on Software Testing, Analysis, and Verification*, pages 36–48, October 1991.

[3] Dirk Grunwald and Harini Srinivasan. Efficient computation of precedence information in parallel programs. In *Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 602–616. Springer-Verlag, 1993.

[4] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[5] G. Kildall. A unified approach to global program optimization. In *Conference Record of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, New York, NY, 1973. ACM.

[6] S. Masticola and B. Ryder. Non-concurrency analysis. In *Proceedings of the Twelvth Symposium on Principles and Practices of Parallel Programming*, pages 129–138, 1993.

[7] S. P. Masticola. *Static detection of deadlocks in polynomial time.* PhD thesis, Rutgers University, 1993.

[8] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering*, pages 24–34, 1998.

[9] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. Technical Report Technical Report 98-44, Department of Computer Science, University of Massachusetts, Amherst, MA, 1998.

[10] J. H. Reif. Data flow analysis of communicating processes. pages 257–268, San Antonio, TX, January 1979.

[11] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[12] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. Technical Report TR-1397, Computer Sciences Department, University of Wisconisin, Madison, WI, March 1999.

[13] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167:131–170, 1996.

[14] M. Sharir and A. Pneuli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice Hall, Englewood Cliffs, NJ, 1981.

[15] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

Appendix: A Formal Definition of the Problem

We present a formal definition of the problem considered in the paper below. The definition extends the traditional concepts of meet-over-all-paths analysis and context-sensitive analysis to define the concept of synchronization-sensitive analysis analogously. The definition is based on the model introduced by Taylor [15] and used by several others. This formalism is based on a control-flow graph representation of procedures, but can easily be adapted for other representations.

Sequential Single-Procedure Programs

We will first consider sequential (*i.e.*, non-concurrent) programs. In the absense of procedures and procedure-calls, a program can be represented by a simple control-flow graph. A single-procedure control-flow graph G is a tuple $\langle V, E, s, e \rangle$ consisting of a set of vertices V, a set of edges $E \subseteq V \times V$, a special "start" vertex s, and a special "exit" vertex e. We assume, without loss of generality, that the start vertex s has no predecessors and that the exit vertex e has no successors. We say that (u, v) is an edge in the graph if $(u, v) \in E$. A sequence of vertices $[u_1, u_1 \cdots, u_k]$ is said to be a path in the graph if for $1 \leq i \leq k$, (u_i, u_{i+1}) is an edge in the graph. We will denote the set of vertices of a single-procedure control-flow graph G by $V(G)$.

Sequential Multi-Procedure Programs

Now consider a (sequential) program consisting of several procedures. We can represent such a program by a collection of single-procedure control-flow graphs, one for every procedure in the program, where some vertices are distinguished as being "call sites". A call site vertex denotes a procedure call, and for such vertices we assume that the called procedure is known.

More formally, a multi-procedure control-flow graph MPG is a tuple $\langle P, \langle G_p : p \in P \rangle, main, \gamma \rangle$ consisting of a set of procedures P, a distinguished procedure *main* belonging to P, a single-procedure control-flow graph G_p for every procedure p in P, as well as a partial function γ from the set $\cup_{p \in P} V(G_p)$ to P. We refer to $\cup_{p \in P} V(G_p)$ as the set of vertices of MPG and denote it by $V(MPG)$. A vertex u in MPG is said to be a call-site if γ is defined for that vertex, in which case we refer to $\gamma(u)$ as the procedure called at call site u. We assume, for the same of simplicity, that call-sites are distinct from the start or exit vertices of any single-procedure control-flow graph. We also assume, for the sake of simplicity, that a call-site has exactly one successor. It is straight forward to ensure that these assumptions are satisfied, by introducing some dummy vertices if necessary. We will refer to the start and exit vertex of the *main* procedure as the start and exit vertex of the program.

Edges in a multi-procedure graph can be classified into two kinds. Edges whose source is not a call-site are *intra-procedural* edges. Edges whose source is a call-site are *return-fall-through* edges. A frequently used alternative representation of a multi-procedure graph is the "supergraph", which is obtained by taking the union of the graphs G_p and replacing every return-fall-through edge $u \to v$ by a pair of labelled edges $u \to s(G_{\gamma(u)})$ (an edge from the call site to the start vertex of the called procedure, referred to as a "call edge") and $e(G_{\gamma(u)}) \to v$ (an edge from the exit vertex of the called procedure to the successor of the call site, referred to as a "return edge"). We attach a label "$(_u$" to the call edge and a label "$)_u$" to the return edge. (Recall that u denotes the call-site.) The labelling on the edges serves to match call edges with the corresponding return edges. A sequence of vertices in a multi-procedure graph is said to be a *supergraph path* if it forms a path in the corresponding supergraph.

Intraprocedural analysis algorithms (*i.e.*, algorithms designed for procedure-less programs) can be adapted to analyze programs with (potentially recursive) procedures by treating a supergraph as a single-procedure control-flow graph. Such algorithms are *context-insensitive* interprocedural analysis algorithms. These algorithms are conservatively safe, since every execution of a multi-procedure program corresponds to a path in the supergraph.

However, not every path in the supergraph corresponds to an execution of the multi-procedure program. One property possessed by a path corresponding to an execution of the program is that it does not contain any mis-matched pair of call and return edge. More formally, a supergraph path is said to be a *same-level valid path* if the sequence of labels on the call and return edges of the path belongs to the language of balanced parentheses generated from the nonterminal *matched* by the following context-free grammar:

$$\begin{array}{rcll} \mathit{matched} & \rightarrow & (_u\ \mathit{matched}\ )_u\ \mathit{matched} & \text{for every call site } u \\ & \mid & \epsilon & \end{array}$$

A supergraph path is said to be an *(interprocedurally) valid path* if the sequence of labels on the call and return edges of the path belongs to the language generated from the nonterminal *valid* by the following context-free grammar:

$$\begin{array}{rcll} \mathit{valid} & \rightarrow & \mathit{valid}\ (_u\ \mathit{matched} & \text{for every call site } u \\ & \mid & \mathit{matched} & \end{array}$$

The precision of interprocedural analysis can be improved by exploiting the fact that only valid paths that are executable. Algorithms that treat the set of interprocedurally valid paths as an approximation to the set of all executable paths and compute the "meet-over-all-valid-paths" solution [14] are referred as *context-sensitive* algorithms. An interprocedurally valid path is also referred to as a *context-sensitive* path sometimes.

Concurrent Programs

We now turn our attention to multi-tasking concurrent programs. We use the model described by Taylor [15], and also used by Naumovich and Avrunin [8]. The programming model is a simple one, where the set of tasks is statically specified. A multi-tasking program consists of a collection of tasks, each task being represented by a multi-procedure graph. The only concurrency primitive we consider is that of synchronization: a task T may invoke an *entry* $S.E$ belonging to a specific task S, in which case the task T waits until the task S *accepts* the entry E. If the task S performs an *accept* of entry E when no other task has invoked the entry $S.E$, it waits until some task invokes the entry. Note that the entry invoked or accepted is statically specified. (Thus, entries are *not* first-class values.) This model corresponds to a simple subset of Ada. (Most concurrent progamming languages provide for several more powerful concurrency primitives. Our undecidability result applies to any language that contains the above features or primitives that can be used to simulate the above features directly.)

We can formalize these ideas as follows. A multi-tasking program is a tuple $\langle T, E, G, \mathit{accepts}, \mathit{invokes}\rangle$, consisting of a set of tasks T, a multi-procedure graph G_τ for every task τ in T, a set of entry names E, a partial function *accepts* from the set of vertices $\cup_{\tau\in T} V(G_\tau)$ to E, and a partial function *invokes* from the set of vertices $\cup_{\tau\in T} V(G_\tau)$ to $T \times E$. Further, for any vertex at most one of *accepts* or *invokes* is defined. A vertex for which *invokes* is defined is said to be an entry invocation vertex, while a vertex for which *accepts* is defined is said to be an entry acceptance vertex. (Note that every entry *belongs* to a particular task. We identify an entry by a pair consisting of the task that owns the entry and an entry name. Thus, different entries belonging to different tasks may have the same name. A task may *invoke* an entry belonging to any task, while a task can *accept* only an entry belonging to itself. This is why we choose to model the function *invokes* as one that returns a pair belonging to $T \times E$, while we model the function *accepts* as one returning only an element of E. An *accepted* entry is implicitly assumed to belong to the *accepting* task.)

We now formalize some concepts related to the execution of a multi-tasking program. An execution of a multi-tasking program corresponds to the execution of all the tasks in the program. Hence, we may represent an execution of the program by a collection $\Pi = (\pi_\tau : \tau \in T)$, where each π_τ is a supergraph path in G_τ. We refer to such a collection of paths as a *potential execution sequence* through the program. We will refer to the potential execution sequence as being context-sensitive if every supergraph path is an interprocedurally valid path. Further, we will say that the sequence begins at the program's start vertex set if the first vertex of every supergraph path is the corresponding task's start vertex.

A potential execution sequence specifies the order in which vertices in a given task are executed, but specifies nothing about the order in which vertices belonging to different tasks are executed. Not every potential execution sequence is a legal execution sequence. In particular, a legal execution sequence has to respect the rendezvous constraints in the program. We formalize these concepts below.

Note that a vertex in a task may execute multiple times. In other words, a path π_τ may consist of multiple occurrences of the same vertex. We will refer to an *occurrence* of a vertex in a path as an *event*.

We will identify an event corresponding to the i-th element of a path π_τ by the pair (π_τ, i). (Recall that a path is just a sequence of vertices.) Formally, let π_τ denote some path in the supergraph of a task τ. We define the set of events in π_τ, $Events(\pi_\tau)$, to be the set $\{(\pi_\tau, i) \mid 1 \leq i \leq |\pi_\tau|\}$, where $|\pi_\tau|$ denotes the length of the sequence π_τ. The set of events in a potential execution sequence $\Pi = (\pi_\tau : \tau \in T)$ is defined to be $\cup_{\tau \in T} Events(\pi_\tau)$ and will be denoted by $Events(\Pi)$.

The sequencing constraints in a multi-tasking program, induced by the rendezvous constructs, constrain the order in which the different events may occur. An event (π, i) is said to be a *synchronization event* if the vertex $\pi(i)$ is either an acceptance vertex or an invocation vertex. (For any sequence π, we will denote the i-th element of the sequence by $\pi(i)$.) We denote the set of all syncronization events in a potential execution sequence Π by $SyncEvents(\Pi)$. A synchronization event (π, i), is said to be a *potential matching event* of another synchronization event (π', i') if the two vertices $\pi(i)$ and $\pi'(i')$ denote an invocation vertex of an entry (τ, e) and an acceptance vertex of the entry e in task τ.

A *potential rendezvous mapping* ξ is a symmetric bijection on the set of all synchronization events of a potential execution sequence that maps every synchronization event to another potential matching event. In other words, ξ is a function from $SyncEvents(\Pi)$ to $SyncEvents(\Pi)$ such that (i) $\xi(e) = e'$ iff $\xi(e') = e$, and (ii) $\xi(e)$ is a potential matching event for e.

For any path π_τ, its *event ordering graph* $D(\pi_\tau)$ is the graph whose set of vertices is $Events(\pi_\tau)$ and whose set of edges is $\{(\pi_\tau, i) \rightarrow (\pi_\tau, i+1) \mid 1 \leq i < |\pi_\tau|\}$.

Given a potential rendezvous mapping ξ for a potential execution sequence $\Pi = (\pi_\tau : \tau \in T)$, we define the event ordering graph $D(\Pi, \xi)$ to consist of the union of the $D(\pi_\tau)$ for every τ in T combined with the set of "synchronization" edges ξ. More precisely, $D(\Pi, \xi)$ consists of the set of vertices $Events(\Pi)$ and the set of edges $\{(\pi_\tau, i) \rightarrow (\pi_\tau, i+1) \mid \pi_\tau \in \Pi, 1 \leq i < |\pi_\tau|\} \cup \{ e \rightarrow \xi(e) \mid e \in SyncEvents(\Pi) \}$.

Thus, $D(\Pi, \xi)$ represents the constraints imposed by ξ on the order in which events of Π can occur. These constraints include the total ordering on the events within a single task, as well as the constraint that two events e and $\xi(e)$ that synchronize with each other must occur simultaneously. The synchronization constraint is modelled by two edges, one from e to $\xi(e)$ and another from $\xi(e)$ to $\xi(\xi(e)) = e$. Thus, for any synchronization event e, e and $\xi(e)$ form a cycle in the graph $D(\Pi, \xi)$. We will refer to such cycles as *matching 2-cycles.* We say that two events e and e' *may happen in parallel* in $D(\Pi, \xi)$ if there exists no path in $D(\Pi, \xi)$ either from e to e' or from e' to e.

A potential rendezvous mapping ξ is said to be *consistent* with the potential execution instance Π if it contains no cycles other than the matching 2-cycles.

A potential execution sequence Π is said to be a *synchronization-sensitive* execution sequence if there exists a potential rendezvous mapping consistent with the execution sequence. (Taylor [15] refers to these sequences as *legal* execution sequences.) Recall that a potential execution sequence is said to be context-sensitive if every supergraph path in the sequence is an interprocedurally valid path.

Definition 3 (The context-sensitive and synchronization-sensitive reachability problem) *Given a multi-tasking program P, a task τ in P, and a vertex u in τ's supergraph, determine if there exists a context-sensitive and synchronization-sensitive execution sequence $(\pi_{\tau'} : \tau' \in T)$ for P, starting from P's start vertex set, in which the last vertex of π_τ is u.*

Taylor defines several other analysis problems for concurrent programs. We say that a vertex u may *possibly (context-sensitively) rendezvous* with a vertex v iff there exists a (context-sensitive) legal execution sequence Π with a consistent rendezvous mapping ξ and $u = \pi(i)$ and $v = \pi'(i')$ and $\xi((\pi, i)) = (\pi', i')$ for some events (π, i) and (π', i')). The goal of the (context-sensitive) "Possible Rendezvous" (or PR) problem is to determine if two given vertices in a concurrent program may possibly (context-sensitively) rendezvous.

Another problem closely related to the PR problem is the "May Happen in Parallel" (or MHP) problem, which is to determine if two given vertices in a concurrent program may ever execute in parallel. We say that a vertex u may *(context-sensitively) happen in parallel* with a vertex v iff there exists a (context-sensitive) legal execution sequence Π with a consistent rendezvous mapping ξ and $u = \pi(i)$ and $v = \pi'(i')$ for some two events (π, i) and (π', i')) that may happen in parallel.

We can, in an analogous fashion, also define generalizations of various analysis problems (*e.g.*, the "reaching definitions" problem) that have been studied in the context of sequential programs.