As concurrent (centralized, parallel, or distributed) software becomes more widespread, the problem of how to validate concurrent programs becomes more critical. Validation of concurrent programs is difficult because of the existence of synchronization between concurrent processes in a concurrent program.

Let \( P \) be a concurrent program. An execution of \( P \) exercises a sequence of synchronization events referred to as a feasible synchronization sequence (or feasible SYN-sequence). (The definition of a SYN-sequence of \( P \) depends on the synchronization constructs used in \( P \).) A synchronization error in \( P \) refers to the existence of a feasible SYN-sequence of \( P \) that is not allowed according to the specification of \( P \). To detect synchronization errors in \( P \), an ideal solution is to derive the set of feasible SYN-sequences of \( P \), called the feasibility set of \( P \), and compare this set with the specification of \( P \). However, this solution has two major problems. One problem is that it is generally impossible to determine the feasibility set of a concurrent program. The other problem is that writing a complete and formal specification of concurrency is often difficult, if not impossible.

One general approach to detecting synchronization errors, called static analysis, is to analyze (not execute) the program to derive an approximation of the feasibility set of a concurrent program. A number of (static) analysis techniques have been developed for detecting synchronization errors [Tay83,Sha85,Avr86]. Generally, these analysis techniques derive an approximation set which is the set of syntactically possible SYN-sequences; such techniques are referred to as syntax-based synchronization analysis techniques. These techniques can detect invalid parallel operations and deadlock. Since the set of syntactically possible SYN-sequences of a concurrent program \( P \) is derived without considering semantic information (e.g., predicates in if-then and loop statements), many syntactically possible SYN-sequences of \( P \) are actually not feasible and many of the errors detected by syntax-based synchronization analysis can never happen. Furthermore, many synchronization errors that result from incorrect predicates in \( P \) cannot be detected by syntax-based synchronization analysis because predicates are ignored.

We have developed a new approach to analyzing concurrent programs, which is to derive constraints on the feasible SYN-sequences of a concurrent program according to the program's syntactic and semantic information. These constraints, called feasibility constraints (or constraints if there is no ambiguity), show restrictions on the ordering of synchronization events allowed by the program. By using feasibility constraints, we can obtain a better approximation of the feasibility set of a concurrent program and improve the effectiveness of error detection by static analysis.

**Definition of Feasibility Constraints**

For a given concurrent language, we can define different types of constraints for different types of restrictions. For example, an execution of a concurrent Ada program exercises a sequence of rendezvous, referred to as a feasible rendezvous sequence (or R-sequence). A constraint for a concurrent Ada program specifies restrictions on the feasible R-sequences of this program. Consider the following constraints, called succession constraints, for a concurrent Ada program \( P \):

1. \([E_1 ; E_2]P\)
2. \([E_1 ; E_2]P\)

The first (second) constraint denotes that during an execution of \( P \), immediately after the occurrence of a rendezvous involving entry \( E_1 \), a rendezvous involving entry \( E_2 \) is always (never) allowed. For task \( R_W\_CONTROL \) in Figure 1, we can derive succession constraints such as:

- \([\text{START}\_\text{READ} ; \text{START}\_\text{READ}] \cdot [\text{START}\_\text{READ} ; \text{START}\_\text{WRITE}] \cdot [\text{START}\_\text{WRITE} ; \text{START}\_\text{READ}]\)

**Derivation of feasibility constraints**

The derivation of constraints for a program
unit U requires the derivation of semantic information in the form of "relations" at various locations in U; each relation describes the relationships (between variables) that hold at the given location. Figure 1 shows several such relations that have been derived in an Ada task called R_W_CONTROL. Task R_W_CONTROL is part of an Ada package that solves the concurrent readers and writers problem using a strategy called "many readers or one writer", which allows multiple readers to access the shared data simultaneously, but only one writer to access the shared data at any time. These relations are shown as comments that begin with "--". The relations derived for synchronization analysis are similar to the assertions derived for program verification. But, the derivation of relations is much simpler.

The syntactic and semantic information of a program unit U can be represented graphically by constructing the semantics-graph of U. Informally, the semantics-graph of a program unit U is the syntactic structure of U with the insertion of the conditions (e.g., predicates in if-then and loop statements) and the derived relations of U. Fig. 2 shows the semantics-graph of an Ada task body T consisting of:

while C0 loop select ... end select; end loop;
Ci, 1<=i<=n, denotes the condition in a guard, Ri the relation at the end of an accept alternative, and R0 the relation before the while loop. (Task R_W_CONTROL is an example of such a task.) Constraints for U are derived from the semantics-graph of U based on the desired types of constraints. After deriving constraints for individual program units of a concurrent program, we can integrate these constraints to produce constraints for the whole program. More details of this semantics-based analysis approach can be found in [Car88].


Figure 1. Task R_W_CONTROL

```
task body R_W_CONTROL is
  NO_READERS:NATURAL:=0;
  WRITER_PRESENT:BOOLEAN:=FALSE;
begin
  -- (NO_READERS=0 and
  -- WRITER_PRESENT=FALSE)
  loop
    select
      when not WRITER_PRESENT =>
        accept START_READ;
        NO_READERS:=NO_READERS + 1;
        -- (NO_READERS>0 and
        -- WRITER_PRESENT=FALSE)
      or
        accept END_READ;
        NO_READERS:=NO_READERS - 1;
        -- (NO_READERS>=0)
      or
        when NO_READERS=0 and not
        WRITER_PRESENT =>
          accept START_WRITE;
          WRITER_PRESENT := TRUE;
          -- (NO_READERS=0 and
          -- WRITER_PRESENT=TRUE)
        or
        accept END_WRITE;
        WRITER_PRESENT := FALSE;
        -- (NO_READERS>=0 and
        -- WRITER_PRESENT=FALSE)
      or
        terminate;
    end select;
  end loop;
end R_W_CONTROL;
```

Figure 2. Semantics-graph of Task Body T