Dealing with concurrent systems poses many interesting and challenging problems. Clearly, it is much harder for developers to reason about concurrent behavior than sequential behavior, and thus it is likely that more errors will be introduced into these systems. Because of this added complexity as well as the difficulties with reproducing results and simulating realistic scenarios, it is important that analysis techniques be developed to evaluate the reliability of concurrent systems. Here we present a representation for concurrent systems, called a task interaction graph, that facilitates analysis.

Our representation is an extension and improvement upon the work of Taylor [Tay83a, Tay83b]. Using a reduced control flow graph representation of each task in a system, Taylor defines a concurrency graph that models the behavior of the total system. Since concurrency graphs capture all the possible states of a concurrent system, they provide an interesting model upon which to base a number of different analyses. For example, using this representation data-flow analysis techniques have been developed to search for specified sequences of suspicious events [Tay86]. Unfortunately the number of states in a concurrency graph can be very large, thereby limiting the programs that can be analyzed and the types of analysis that can be performed.

We have been developing a model of interacting tasks that may considerably reduce the number of states in concurrency graph representations. We call this representation a Task Interaction Concurrency Graph (TIG), since it is derived from a Task Interaction Graph (TIG) instead of from a control flow representation. Using our model, we compared the resulting representations for some of the common concurrency examples that appear in the literature. For these examples the number of states were reduced by well over fifty percent. Such a substantial reduction will have a major impact on the kinds of analysis that can be applied and on the kinds of programs that can be analyzed. Moreover, this reduction comes with no loss of information. In fact, we believe that our resulting representation is even more amenable to analysis. This is because a TIG divides a task, not based on control flow information, but based on task interactions, the real focus of our concern. The other benefit of this choice of representation is that the nodes in a TIG identify maximal sequential regions in the task. Thus, sequential analysis techniques could be applied to these regions and then inter-task analysis techniques developed to evaluate the impact of task interactions, in much the same way that inter-procedural analysis is carried out for program optimization.

The TIG and TIG models have been designed to capture the rendezvous-like synchronization found in languages like Ada [Geha83], Distributed Processes [Brin78], and CSP [Hoar78]. Our work to date uses an Ada-like language, but the approach is applicable to other languages that use a rendezvous model of synchronization.

Task interaction graphs represent tasks as sets of regions and interactions between regions. Formally, a task interaction graph is a tuple \((N, E, S, T, L, C)\), where \(N\) is the set of nodes, \(E\) is the set of edges, \(S\) is the start node, \(T\) is a set of terminal nodes, \(L\) is a function that assigns a label to each edge, and \(C\) is a function that assigns pseudocode to each node. Each node of this graph represents a task region and each edge represents a task interaction. The start node represents the region where the task starts execution. The terminal nodes represent regions where the task may finish execution.

Each node in a TIG represents a different region of the task and has associated with it an explicit representation of the code for that region, referred to as pseudocode. The pseudocode for regions can be any convenient representation augmented by the addition of two transition pseudostatements that mark the beginning and the end of regions.

Each edge in a TIG represents a task interaction, indicating a transition from one region to another. The boundary between these regions is represented by the two transition pseudostatements - one in each of the two regions connected by that interaction. The pseudostatement in the first region indicates a place where that region may be exited, and the pseudostatement in the second region indicates the place where that region may be entered.

In the rendezvous model a task makes an entry call to a task, which in turn accepts the call. Each call and each accept statement is modeled using two interactions that divide the task into three regions. Calls and accepts are divided into two interactions each (e.g., starting an entry, ending an entry, starting an accept, ending an accept) because when a rendezvous is initiated, information can be passed from the calling task to the accepting task via the parameters of the call and accept statements. This changes the environment of the accepting task, dividing it into two regions at this point. When the rendezvous is ended, information can be passed in the other direction, dividing the calling task into two regions at this point. However, there are special cases where a more compact representation can be used.

To date we have developed rules for translating most of the constructs supported by Ada into the appropriate TIG represen-
for the toy examples in the literature that we have examined so far, the number of states in the TICG has been relatively small. Furthermore, each resulting TICG has been substantially smaller than the corresponding control flow concurrency graph. Our hypothesis is that the complexity of the TICG for the typical system will be quite reasonable, although worse case analysis clearly shows it is an intractable problem [Tay83a]. We feel it is imperative to conduct some experimental studies so we can evaluate typical performance for realistic systems. We are currently building a prototype system that can automatically create the TIG and TICG for actual, production programs. We intend to use the prototype to do experimental studies on the size and complexity of the generated graphs.

We have been investigating several kinds of analysis techniques that can be applied to the TIG and TICG models. Some of these techniques are relatively simple to apply and can be carried out during the creation of the graphs, and others require post processing, which might even be directed by information gathered during the creation of the graphs. An example of one kind of analysis that can be done during the creation of the TICG is deadlock detection. An example of the kind of analysis that requires post processing is “dangerous” parallelism. This occurs when a global variable can be assigned and referenced in various orders. The possibility of such situations can be detected during TICG construction and then analyzed afterwards.

Deadlock detection and dangerous parallelism are just two examples of the kinds of analysis that can be performed using a TICG representation. We believe the model is amenable to more extensive and powerful analysis techniques. Because sequential processing is carefully separated from task interactions, it appears that some sequential analysis techniques could be applied to task regions and the results incorporated into inter-region analysis, similar to the techniques currently used for inter-procedural analysis.

We are particularly interested in investigating the extension of error sensitive testing techniques to concurrent, real-time systems. In fact, it was this very problem that led to the development of the TIG and TICG models. We are particularly interested in extending the RELAY model [Rich87] since this appears to be a general and powerful technique for describing criteria that must be satisfied in order to reveal certain classes of faults. Based on our initial investigations, it appears that task interaction graphs and concurrency graphs will be very useful toward that goal.

References


