Abstract
Task graph representations of complex software systems often require the specification of multiple constraints upon the execution of a single task. In this paper, existing formalisms associated with this type of specification and their underlying execution models are discussed. These formalisms are analyzed from different structural levels of the software system to identify their limitations in describing systems having client-server relationships. In these types of software systems, the combination of the precedence constraints must support the dynamic job structure of the system. Existing formalisms do not offer this support while accurately representing the desired behavior of the software.

Key Words: precedence constraints; software scheduling; constraint combination; task graph representations

Introduction
A software task is user-generated software that forms an individually schedulable entity on the targeted host computing system. It is sometimes referred to as a software process, and the terms task and process are synonymous in this paper. Tasks can exist in one of three possible states, running, ready, or blocked. Each instantiation of a task is called a job. Tasks can have one associated job or a series of associated jobs. If a task spawns a series of jobs that are instantiated at regular time intervals, then the task is executing periodically. If there is a single job or if the series of jobs of a task are not instantiated at regular, predictable time intervals, the task is said to be executing aperiodically. This is a classic description of a software task [1].

Most complex software systems are composed of a set of cooperating tasks. These tasks often have precedence relationships resulting from dependencies among the tasks in the set. These relationships are manifested in the form of synchronizations and communications among the tasks and define a partial order of execution for the set of tasks. This partial order must represent the characteristics of the software system, and it must not be altered by the scheduling algorithm used on the host computer.

For a task having a single dependency upon one other task in the system, the partial order of execution is easily determined. In this case, the task having the dependency, called the dependent task, cannot enter the ready queue until the task upon which it is dependent, called the predecessor task, has completed execution. Figure 1 shows a task graph representation for a task having a dependency upon one other task. In the task graph, the software tasks are represented as nodes of the graph and the dependencies among the tasks are represented as directed arcs connecting the nodes. The tasks are given numeric labels, and the arcs are assigned a label and an ordered pair representation indicating the source node and the destination node for each arc. The execution timeline demonstrating the partial order defined by this example is also shown.

![Figure 1](image_url)
Creating a partial order of execution for a task set becomes more difficult when one task has dependencies upon two or more tasks in the set. In this case, semantics must be defined that clearly describe how these multiple constraints are to be combined to determine the partial order of execution of the tasks involved. For example, consider the task set shown in Figure 2. The task graph represents a five-node graph structure in which there are four predecessor tasks that communicate with a single dependent task.

The set of all incident arcs to task 5 is denoted by $I(5)$, where

$$I(5) = (a1, a2, a3, a4).$$

The cardinality of set $I(5)$ indicates that there are a total of four constraints that in some way govern the execution of task 5. The next section describes the existing formalisms defining exactly how these constraints are combined to form a partial order of execution for the set of tasks.

**Existing Constraint Combination Methods**

There are several existing methods that can be used to combine multiple constraints on the execution of a single task. Each constraint is based upon the need for synchronization between tasks. A constraint is satisfied when the dependent task actually receives the synchronization mechanism from its predecessor. Worst-case execution characteristics are considered by assuming that all synchronizations are sent at the end of each predecessor job. This serializes the execution of the dependent task and its predecessor, allowing for less parallelism in the schedule. It is assumed throughout this paper that a constraint that has been satisfied has a logical value of TRUE.

Existing constraint combination methods are based upon performing a logical operation on the set of constraints. For example, the dependent task may become ready for execution only after all of its constraints have been satisfied. In effect, a logical AND operation is performed on the set of constraints. This operation can be represented in a task graph using a special notation for such an AND task [2]. The same relationship is sometimes represented by describing the incoming arcs as AND arcs [3]. Still other task graph descriptions refer to the tasks that are adjacent from the dependent process as Parallel AND jobs [4] or AND-task graphs [5]. In this paper, and in other work [3][6], this is referred to as AND-join semantics. Although the names and the graphical representations are different, the semantics of the combination of the dependencies remain the same. The dependent task becomes ready for execution only after all of its constraints have been satisfied, which corresponds to the completion of all the predecessor tasks. For the example shown in Figure 2, this method of combining the constraints in set $I(5)$ is shown as

$$5_{rs} = a1 \land a2 \land a3 \land a4,$$

where $5_{rs}$ is a logical value representing the ready status of task 5. It is assumed throughout this paper that higher task numbers represent tasks with higher priorities. As a result, a priority-based scheduler will produce the execution order shown in Figure 3 for the task set in Figure 2 using AND-join semantics.

A second method of determining the ready status of a task having multiple dependencies involves performing a logical OR operation on the set of constraints. Specifically, this means that the dependent task is ready to execute after any one of its constraints is satisfied, corresponding to the completion of any one predecessor task. Again, there are many different descriptions of this relationship in a task graph. The OR operation can be assigned as a property of the dependent task [2], or it can be described as part of the arc [3]. The predecessors are sometimes grouped and described as an OR-task graph [5], or a Parallel OR job [4]. In this paper, as in other work, we will refer to this model as OR-join semantics [3][6]. For the example shown in Figure 2, this method of combining the constraints in set $I(5)$ is shown as

$$5_{rs} = a1 \lor a2 \lor a3 \lor a4,$$

and the corresponding partial order of execution is shown in Figure 4.
There are a few variations associated with the OR-join semantics. First, a threshold can be specified representing the number of constraints that must be satisfied before the dependent task can execute. The threshold must be greater than or equal to one and less than the total number of constraints on the task. For example, task 5 in Figure 2 may become ready to execute after any two of its constraints are satisfied. The logical operation associated with this example is

$$5_{21} = (d \land a2) \lor (d \land a3) \lor (a2 \land a3) \lor (a2 \land a4) \lor (a3 \land a4).$$

(4)

Such threshold graphs have applications in fault tolerant systems and represent a more general case of the traditional OR-join semantics having a threshold of one [3][6]. If the threshold equals the number of constraints on the task, then such a specification would be equivalent to that of AND-join semantics. Figure 5 shows the execution order for the task set in Figure 2 assuming OR-join semantics with a threshold of two and the priority-based scheduling environment.

Figure 5. The execution order for the tasks in Figure 2 assuming OR-join semantics with a threshold of two.

A second issue associated with OR-join semantics introduces even more variations. This issue deals with how the tasks that do not actually trigger the ready status of the dependent task are to be treated in the model. For example, consider the task graph in Figure 2 and assume that OR-join semantics with a threshold of one are employed. In this example, task 5 becomes ready to execute on the completion of tasks 1, 2, 3, or 4 whichever completes first. Assume task 4 completes first. The execution of tasks 1, 2, and 3 can be handled in different ways leading to several different execution models associated with the OR-join semantics. The predecessor task that completes execution first and triggers the ready status of the dependent task, task 4 in the above example, is called an essential task. The other tasks, tasks 1, 2, and 3 in the above example, are called inessential tasks [2].

There is an execution model, called the skipped problem, in which the inessential tasks are either not scheduled at all, or their execution is immediately terminated upon the completion of the essential task [2][4]. Figure 6 shows the execution timeline for such semantics assuming the inessential tasks are not scheduled for execution.

![Figure 6](image)

Figure 6. The execution order for the tasks in Figure 2 assuming skipped OR-join semantics with a threshold of one.

A second execution model exists, called the unskipped problem, in which the inessential tasks must be scheduled and executed [3]. Figure 4 shows the unskipped semantics for the same task set executed in Figure 6. Also, Figure 5 shows the unskipped model using the OR-join semantics with a threshold of two.

**Task and Job Structures of Complex Software Systems**

In a deterministic environment, the structure of a concurrent task system can often be determined statically before run time. This is because the structure can be inferred directly from the program code and every process involved in the software system exists as long as the software system itself exists. As a result, one task graph can represent the entire software system at any point during its execution. Such a task graph structure in itself does not fully describe the intended behavior of some systems. At a level of abstraction closer to the execution specifics of the system, the software system can also be described according to its job structure. Recall that the term job refers to a single instantiation of a task, and a single task may have one or many associated jobs. If all tasks in the system have only one associated job, then the job structure of the system is the same as the task structure. If one or more tasks in the system spawn a series of jobs, the job structure becomes dynamic. The job structure evolves during the execution of the system and represents the execution of the system at a particular instant of time. In these systems, the task and job structures are different, but both are needed to fully describe the behavior of the system. The task structure represents the high-level structure associated with the software system, and the job structure gives the low-level details associated with the execution of individual instantiations of each task and how these instantiations interact in a dynamic manner with other jobs [7].

Existing constraint combination formalisms apply a logical operation on the task structure of the system. As a result, they accurately describe systems having identical task and job structures. However, these formalisms do not apply to software systems having dynamic job structures. These formalisms are unable to accurately represent the desired behavior of such systems. Systems with client-server relationships are examples of these types of systems [7].
Limitations of Existing Methods for Client-Server Systems

Consider applications having client-server relationships. Clients are defined as user-generated tasks that request services from other user-generated tasks. The server is also a user-generated task that provides services to clients. The server must respond to the requests from each client and can begin servicing requests as soon as a request is received. One task can act as both a client and a server by requesting a service from one task and providing a service to a different task.

Functionally, the client is dependent upon the server for some service. However, from an execution ordering perspective, the server is dependent upon the client. The server must wait until the client executes and requests a service before it can service the request. It is this dependency that is shown in a task graph by an arc leading from the client to the server, and this dependency represents a constraint upon the execution of the server task. When a server is providing services to many clients, there are many constraints on the execution of the server task requiring semantics for the combination of these constraints.

As an example consider the task graph in Figure 7. In this graph, task 3 represents a server for the two clients, tasks 1 and 2. For this example, it is assumed that each client has only one associated job. This results in one request for service from each client during the execution of the entire software system. The set of constraints upon the execution of task 3 is

\[ I(3) = (a_1, a_2). \]  

(5)

Applying the unskipped OR-join semantics with a threshold of one to set \( I(3) \) in Equation 5 and maintaining the assumptions regarding the priority-based scheduling environment discussed above results in the execution order shown in Figure 7.

![Figure 7. Task graph and execution order for a client-server system assuming the unskipped OR-join semantics with a threshold of one.](image)

Another problem with the application of these semantics to systems having client-server relationships is that there is still no support for the creation of multiple server jobs to service all the requests. The ready status of the server is still calculated by applying a logical operation to the entire set of constraints as defined at the task level, set \( I(3) \) in Equation 5. The constraints are not considered as individual requests for service, and the job structure of the server task is not considered. As a result, only a single server job is instantiated [7].

Since all the requests are being made before any of them are serviced, there are issues related to which request is actually getting serviced when the server finally does execute. The request that is serviced at this time is implementation dependent. One possibility is that the
first request that was issued to the server will be the only request serviced. In this case, this pending request blocks all the other clients' requests. A second possibility is that the server will only service the last request received from a client. In effect, each client request overwrites the previous request that was still pending. One might be inclined to assume that a mechanism used to queue the client requests would solve some of these issues. However, the requirement that requests are queued is not part of the basic client-server relationship specification. Even with the addition of the queue, the problem of dynamic creation of server jobs to service requests in the queue is not addressed [7].

**Limitations of Existing Formalisms for Client-Server Systems Having Periodic Workloads**

Applications having periodic workloads present further difficulties for existing formalisms. The periodic tasks within the system share a timing relationship based upon major and minor frames. The major frame, also called a hyperperiod or a master period, is the least common multiple of the frequencies of the periodic tasks and represents the longest period among all the periodic tasks in the system [6][8]. The minor frame is the shortest period among all the periodic tasks and represents the fundamental unit of time [6][8]. Most of these systems impose the limitation that the frequencies of all periodic tasks must be harmonics, making it easy to decompose the major frame into an integer number of minor frames. As the system executes, the execution characteristics of all periodic tasks repeat every major frame. But, execution characteristics for the entire system are not repeated every minor frame, which is the fundamental time unit. Some periodic tasks will execute in some minor frames but not in others. All the periodic tasks may not have the same number of jobs that execute during each major frame. The task and job structures of such systems are completely different, and the point at which a logical operation is applied to the set of constraints, even the constraints as defined at the task level, is not defined.

Consider the task set and the execution characteristics defined in Figure 9. These execution characteristics are defined by the software architect and represent three clients requesting service from one server task. The clients execute as periodic tasks, each spawning a series of jobs and each job requesting service from the server. All the frequencies are harmonics, and the major frame time is ten milliseconds and the minor frame time is five milliseconds with two minor frames per major frame. Task 4 is aperiodic since it is impossible to predict its timing from that of the clients.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Frequency</th>
<th>Minor Frames</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>200 Hz</td>
<td>1 and 2</td>
<td>2 ms</td>
</tr>
<tr>
<td>2</td>
<td>100 Hz</td>
<td>1</td>
<td>1 ms</td>
</tr>
<tr>
<td>3</td>
<td>100 Hz</td>
<td>2</td>
<td>1 ms</td>
</tr>
<tr>
<td>4</td>
<td>aperiodic</td>
<td>not applicable</td>
<td>1 ms</td>
</tr>
</tbody>
</table>

Figure 9. The execution characteristics of the periodic clients and the server.

Figure 10 shows the execution order for the clients. Higher task numbers still represent higher execution priorities which are used to determine the execution order for tasks in the ready queue.

The job structure of the system is completely different than the task structure. The constraints upon the execution of the server are not completely represented at the task level. The point at which the logical operations associated with the traditional semantics are applied to the set of constraints to determine the correct combination of the constraints is undefined. It could be applied once within each minor frame or once within each major frame. If it is applied once during each major frame, processes that execute multiple times during the major frame become an issue. If it is applied once during each minor frame, processes that do not execute at all during that minor frame must be accounted for in some way. They did not execute within that particular minor frame, but the execution of the server is still dependent upon their execution in some fashion. Similarly, it is uncertain if the server is triggered for execution just once, or a series of server jobs is created somehow associated with the series of jobs from the clients. So, it is easy to see how the dynamic job structure of such an example creates problems for the existing semantics. Traditional semantics are not defined for such applications [7].

**Conclusions and Future Research**

The traditional OR-join and AND-join semantics are not applicable to software systems having client-server relationships since these systems have a dynamic job structure that is not represented by the static task structure, the level at which these semantics are typically applied. The existing semantics do not support the dynamic creation of server jobs to handle multiple service requests and do not accurately represent the desired
behavior of the system. If the software system has periodic clients, the job structure becomes based upon major and minor frame timing relationships that are not represented at all in the task structure of the system. The application of the traditional semantics to such systems is not defined.

Currently, work is being done to develop constraint combination semantics that can represent the behavior of these types of systems. Called the Modified OR-join semantics, these semantics show potential for application at the task level while representing the underlying dynamic job structure of client-server systems [7].

References


