Preconditions, Postconditions, and Provisional Execution in Software Processes

Stanley M. Sutton, Jr.

CMPSCI Technical Report 95–77
July 1995

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the Advanced Research Projects Agency under grant F30602-94-C-0137.

Preconditions, Postconditions, and Provisional Execution in Software Processes

Stanley M. Sutton, Jr.

CMPSCI Technical Report 95–77
July 1995

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

This work was supported by the Advanced Research Projects Agency under grant F30602-94-C-0137.

Abstract

Preconditions and postconditions are widely used in execution models for software processes. The conditions that control software processes, however, can be complex and difficult to evaluate in the context of ongoing development activities. If evaluations and tasks are conducted in parallel, evaluations may be blocked for lack of access to data that are held by tasks. Conversely, task initiations and terminations may be blocked because their controlling conditions cannot be evaluated. Costly delays or even deadlocks may result. Potential conflicts between condition evaluations and process tasks can be reduced by a number of strategies, such as limiting the extent of conditions, or serializing evaluations and tasks. However, each of these strategies has its drawbacks (such as failure to capture full process semantics, or reduced process concurrency).

This paper introduces a new strategy for condition-based process execution, specifically, provisional execution of process tasks. When it is not possible to completely evaluate all preconditions or postconditions because of conflicts with ongoing activities, a manager may allow some tasks to be executed provisionally. That is, a task may be initiated before all its preconditions are satisfied, or terminated before all its postconditions are satisfied. The blocked conditions are not ignored; rather, their evaluation is deferred but still required. These conditions must be satisfied or the task is subject to abort. Provisional execution allows increased flexibility in both the execution of process tasks and the evaluation of controlling conditions. It can lead to increased concurrency, reduced concurrency conflicts, and generally greater adaptability in process execution.

1 Introduction

Preconditions and postconditions are widely used in execution models for software processes [17, 6, 24, 25, 16, 15]. These conditions typically characterize process or product state and allow process execution to be controlled accordingly. For example, preconditions may govern which tasks are selected or allowed to execute, while postconditions may govern which tasks are permitted to commit or otherwise contribute to a process. The use of preconditions and postconditions in software process execution has several benefits. It makes important aspects of process and product semantics clear, it captures interrelationships between state and behavior, and it allows alternative control paths to be easily expressed.

The conditions that control software processes, however, may be very complex and difficult to evaluate in the context of ongoing software development activities. The evaluation of conditions may require long-term access to widespread data that are likewise needed for extended periods by process tasks. This raises the prospect that process tasks and condition evaluations may conflict over concurrent access to data they both require. This may lead to a breakdown in process execution if either process tasks or condition evaluations need to be delayed or possibly even aborted. Potential conflicts between condition evaluations and process tasks can be reduced by a number of strategies, such as limiting the extent of conditions, or serializing evaluations and tasks. Each of these strategies has its drawbacks, though (such as failure to capture full process or product semantics, or reduction of process concurrency).

This paper proposes a new strategy for process execution based on preconditions and postconditions, that is, provisional execution of process tasks. Provisional execution is based on delayed evaluations of preconditions and postconditions. More specifically, under managerial control, task execution may be partly decoupled from condition evaluation. This can result in greater flexibility for both the execution of process tasks and the evaluation of controlling conditions, in particular leading to increased concurrency and reduced concurrency conflicts.

An overview of condition-based execution models for software processes is presented in Section 2. The concept of condition-based execution is explained in Section 2.1, the problems that may arise for condition based execution are explained in Section 2.2, and a comparison with the related problem of long transactions is made in Section 2.3. Strategies that mitigate the problems of conditional execution are reviewed in Section 3. The alternative strategy of provisional execution for process tasks is described in Section 4. A model of task execution that incorporates provisional states is presented in Section 5; some issues of process modeling and control are discussed in Section 6. Related work is surveyed in Section 7, and the results are summarized in Section 8.

2 Condition-Based Execution

This section describes condition-based execution, indicates the concurrency problems that may arise from it, and compares these problems to the related problem of long transactions.

2.1 The Concept

Preconditions and postconditions are widely used in software process models, languages, and environments. Preconditions and postconditions can be defined generally as conditions that are evaluated before or (respectively) after an associated operation, where the result of the condition evaluation has some consequence with respect to the execution of the operation. Execution models that incorporate preconditions and postconditions may be called *condition-based.*

A large variety of condition-based execution model have been developed. Merlin [16] has consistency conditions and automation conditions. Both assert changes to object state (i.e., status) as a postcondition effect for completed action, but either kind of condition can lead to the triggering of a corresponding kind of transaction (for which the condition serves as a precondition). Marvel [19] rules have a property list that must be satisfied by the given arguments before the rule can be fired. These property lists act as guarding preconditions on user-invoked rules and as triggering preconditions on automation rule chains. Marvel also has consistency rule chains that act like constraining postconditions on rule results (i.e., failure to assert the postcondition effect of an action leads to rollback of the action). AP5 [6] has automation rules that are fired by triggering preconditions and consistency rules that are fired by constraining postconditions. Violation of the condition for a consistency rule can lead first to the firing of a repair action, but failure to satisfy the condition ultimately leads to rollback of the original action. EPOS [7] has both static and dynamic preconditions and postconditions. The static conditions are used for planning, the dynamic conditions are used for triggering actions or reactions. Pleiades [25] provides a constraint construct that may be evaluated as a precondition or postcondition to operations on instances of abstract data types. As preconditions these constraints can serve guards, as postconditions they can serve as constraints, and in either case they may serve to trigger operations that are supplemental to the invoked operation. APPL/A [22, 24] has predicates that can be enforced as constraining postconditions on relation operations. SLANG [2] and Melmac [8] have what might be called selecting preconditions; these are guards on alternative execution paths which are used to control branching execution. Grapple [15] makes use of postcondition goals in process planning and recognition.

In these systems (and others) the most common use of preconditions and postconditions is to control or otherwise affect process execution. Most typically, preconditions in some way determine whether a task is or may be executed, while postconditions in some way determine whether the results of a task are committed or otherwise judged acceptable for some purpose.

Another common characteristic of these systems, one that is most important to this work, is that preconditions and postconditions are evaluated by the process execution engine (which may include the language interpreter or runtime system). For example, in Marvel, preconditions and postconditions are evaluated by the Rule Processor (part of the Server in Marvel), in AP5 they are evaluated by the AP5 interpreter, while in APPL/A and Pleiades they are evaluated by the language runtime system, and in SLANG they are evaluated by the SLANG interpreter. In any of these cases, the ability to evaluate the conditions is essential to the proper functioning of the execution engine of which they are part.

2.2 The Problem

Condition-based execution models afford several benefits for software processes. These benefits may include indirect invocation, declarative expression, explicit consistency management, and the ability to combine proactive and reactive control. However, the unrestricted use of condition-based execution for software processes can lead to concurrency conflicts between process tasks and the process execution engine. This can occur if the same data are required both for process tasks and the evaluation of preconditions and postconditions. If evaluations and tasks are conducted in parallel, then delays or deadlocks may result from conflicting accesses to data. Condition evaluations may be blocked for lack of access to data that are held by tasks, and task initiations and terminations may be blocked because their controlling conditions cannot be evaluated.

The problem is potentially acute for software processes because process tasks can be long, complicated, and costly. Additionally, efficient software development often requires that many tasks go on in parallel. In a condition-based execution model, a large number of process tasks implies a concomitantly large number of condition evaluations. Furthermore, the preconditions and postconditions associated with a task may reference data that extend beyond those that are used by the task. For example, a task to unit-test a source-code module may depend on the semantic consistency of the module; the semantic consistency of one module, though, may depend on the semantic consistency of several others. Consequently, the potential for concurrency conflicts may be greater for the preconditions and postconditions associated with the task than they are for the task itself.

Due to the potential cost of process tasks, it is undesirable to delay or abort these in the

event of a concurrency conflict. However, condition evaluations may also be expensive, and if these are aborted or delayed, so is the progress of the tasks that are dependent on them. If, in the event of a conflict, both tasks and evaluations are allowed to proceed, then another potential consequence is interference between them. This may result in incorrect evaluations, which in turn can lead to incorrect decisions in the control of process execution (including, for example, the abort of tasks that should be committed, or the initiation of tasks that should not be started).

2.3 Long Transactions

Some of the relevant properties of software processes give rise to the well-known problem of long transactions in software processes [4]. The problems of long transactions and conditional execution are similar in that both involve concurrency-control issues and both may entail costly transaction aborts. However, these problems are different in several respects.

With long transactions, concurrency conflicts occur between process tasks, but they do not involve the process engine (e.g., deadlock of the transaction manager is not an issue). The conflicts occur over data that are specifically referenced by the transactions involved; additionally, the consequences of conflicts (e.g., delay, abort) immediately affect just those tasks involved in the conflict.

In contrast, with condition-based execution, conflicts occur not just between process tasks but between process tasks and the process engine, which must evaluate conditions in support of task execution. (In this respect, the problem is similar to that of "extra-data" transactions, which are discussed in Section 7.) These conflicts occur not only over data referenced by the process tasks but also over data that are referenced by the conditions that apply to those tasks. The conflicts may adversely affect the performance of the execution engine, since it may be delayed or blocked in trying to evaluate these conditions. As a consequence, tasks other than those involved in the conflict may be affected, since the ability of the engine to effect the execution of those other tasks may be compromised.

3 Strategies for Reducing Conflicts

The problem of concurrency conflicts in condition-based process execution models can be attacked using a variety of strategies. Each of these can mitigate the problem to some extent, but none is entirely satisfactory.

One strategy is to use fewer, narrower preconditions and postconditions. This will tend to reduce conflicts by reducing the number and extent of condition-required accesses to data that may also be in use (or needed by) process tasks. The corresponding limitation of this approach is that control decisions will be based on relatively local conditions while the problems of global control and consistency will remain to be addressed.

A related strategy is to make less use of condition-based execution. Alternatives would be, for example, to make more use of imperative or event-triggered (as opposed to condition-triggered) control models. These execution models certainly seem to have a place in software processes [24, 5]. However, they do not, in and of themselves, address the persistent need for conditional execution, consistency management, and so on. Conditions that might otherwise be expressed separately as preconditions and postconditions must instead be incorporated explicitly into process tasks, and the tasks must take on the burden of evaluating and responding to the conditions.

A different sort of strategy is to attempt to represent software processes using shorter (i.e., less expensive) tasks. In this case, concurrency conflicts are admitted, but they may be more readily resolved by aborting a task since the cost of re-doing tasks is reduced. However, the scale of large software projects, the computational complexity of many development activities, and the inherent need for manual support, imply that long transactions cannot be entirely avoided. The transaction model proposed in [16] addresses this situation by providing both optimistic concurrency-control for transaction that may be more readily aborted and pessimistic concurrency-control for transactions that should be protected against aborts. This may be viewed as a special case of semantics-based concurrency control [3], which can be applied at least to resolve concurrency conflicts in favor of more expensive transactions.

Yet another approach is to attempt to reduce the incidence of conflicts by serialization of process tasks and condition evaluations. Serialization is used, for example, in AP5 [6]. User tasks are subject to consistency rules that are enforced as postconditions. When a user task completes, consistency rules are evaluated and repair actions are taken, before another user task is allowed. This ensures that condition evaluation occurs free from interference by ongoing application activities. Such serialization can be used to protect condition evaluations from interference by process tasks, but it has the direct consequence of reducing parallelism in development activities.

A different approach to reducing the incidence of conflicts is to partition the data spaces in which condition evaluations and process tasks are conducted. One variant on this is the segregation of product state from product attributes, as practiced, for example, in Marvel [17] and Merlin [16]. In this approach, product artifacts are represented as files (or in some external object manager), and process tasks operate on these using the usual sorts of tools.

In contrast, the attributes of the artifacts are stored separately by the process engine where they are not accessible to process tasks. Thus a source code module may be stored as a file but have associated attributes "complied" and "analyzed" that are managed by the process engine. Process tasks operate directly on products, but process control is based on the evaluation of conditions over the derived attributes. This may eliminate direct conflicts between process tasks and the process engine, but the potential for indirect conflicts remains, since the process engine may still need to invoke process tasks to derive attribute values that are needed to satisfy a precondition or postcondition.

Another variant is the use of segregated workspaces for process tasks. Workspaces are used in a number of systems (including some that do not support automatic evaluation of preconditions or postconditions) [5, 7, 16, 10]. Conditions on product state can be evaluated in a central repository while updates proceed in parallel on local copies of the product. Condition evaluations within a workspace can be coordinated with the local task without holding up tasks in other workspaces. Synchronization of updates and evaluations in the central repository can be limited to the periods of check-in, which at least may be brief compared to process tasks. Complications arise in the workspace model, though, from the need to manage replication of the product, including, for example, merging local updates into the central repository, propagating changes between workspaces, and coordinating parallel tasks in separate workspaces.

Each of the strategies described above affords some reduction in the problems associated with condition-based execution of software processes. However, each entails other limitations or complications. Provisional execution of process tasks is recommended here as an additional strategy, one that attempts to retain both condition-based execution and a high level of concurrency. The cost is some risk to process tasks, but these risks may be managed.

4 Provisional Execution

Provisional execution of software processes is based on two premises:

- Since preconditions and postconditions may not always be testable, we should recognize that as a practical matter their state may be *indeterminate*
- Since indeterminacy may be unavoidable, we should define process execution models *accommodate indeterminacy*

Provisional execution of process tasks is intended to accommodate indeterminacy by *relaxing* the execution semantics of preconditions and postconditions.

With respect to preconditions, the basic idea of provisional execution is that a task may be allowed to begin executing before all of its preconditions are known to be satisfied. In general, the provisional execution of a task should not be allowed to proceed beyond some critical point unless the preconditions are satisfied. Provisional execution can be based on the expectation that the preconditions will be found satisfied when needed, or that the potential benefits of early execution outweigh the potential costs of abort if the preconditions are not found satisfied.

With respect to postconditions, the basic idea of provisional execution is that a task may be allowed to complete or commit tentatively before all of its postconditions are know to be satisfied. In general, the results of the task should not be used for critical purposes until and unless the postconditions are satisfied. As with the provisional initiation of a task, the provisional termination of a task can be based on the expectation that the postconditions will be found satisfied, or more generally on the judgement that the potential benefits outweigh the potential costs.

This basic model of provisional execution makes several presumptions. First, it requires runtime support to decouple condition evaluation from task execution. This requires keeping track of dependencies between tasks and conditions and monitoring the corresponding status of tasks. However, task status is already maintained in many process systems, and the tracking task-condition dependencies can be viewed as an extension of the existing need to track inter-task dependencies. Second, it presumes that at least some process tasks can be designed such that some "head start" is useful. This seems reasonable since many tasks involve start-up activities and generally may include aspects to which the preconditions are not relevant. Additionally, many tasks may execute for an extended period before their cumulative costs make aborting the tasks problematic. A third presumption is that it is possible to recognize points in the execution of a task where preconditions become relevant. (Correspondingly, it should be possible to recognize later tasks to which the satisfaction of postconditions by earlier tasks is critical.) At least some such points will be reflected by explicit control or data dependencies, or safety-critical steps, in process specifications. These may be found by manual or automated analysis of the specifications. Other points may not be reflected in process specifications but may be identified by process participants or managers based on their judgement of the state and progress of the task. Finally, this model of provisional execution presumes that the execution of a task can be delayed or revoked (or, in the case of postconditions, that the results of a task can be aborted or discarded). This may be feasible for "lightweight" tasks or for more expensive tasks that are still in their relatively early stages.

Provisional execution offers several potential advantages. By breaking the strict synchronization of precondition and postcondition evaluations with process tasks, it allows greater flexibility in the scheduling of condition evaluations so as to avoid conflicts with ongoing tasks. It can increase concurrency by allowing preconditions evaluations to occur in parallel with the associated tasks (and postcondition evaluations to go on in parallel with dependent tasks). It also allows for the exercise of managerial judgement as to when, where, and how preconditions and postconditions are relevant to ongoing and prospective tasks. (This would be analogous to dynamic, semantics-based concurrency control.) In general, it offers the prospect of greater flexibility in controlling the execution of process tasks while at the same time reducing the potential for conflicts arising from the strict interpretation of condition-based execution models.

5 A Model of Provisional Task Execution

A state-transition model that incorporates provisional states into the life cycle of a process task is shown in Figure 1. This model reflects a number of fundamental aspects of provisional condition-based execution.

The life cycle of a process task is presumed to begin when the task is first identified for execution. At that time, the evaluation of relevant preconditions can begin; until their results are known, the task is regarded as not qualified. (If a task remains not qualified long enough, it may be withdrawn, terminating the life cycle.) If the preconditions are found to be satisfied, the task is considered qualified, that is, ready for execution.[1] Alternatively, if some preconditions are indeterminate but are not considered essential to the invocation or early stages of the task, then the task may enter the provisionally-qualified state.

An important aspect of provisional execution becomes relevant at this point in the task life-cycle. Provisional execution carries with it the implication that preconditions and postconditions can be evaluated over a period of time. More specifically, their evaluation is not strictly synchronized, either among themselves or with the associated task. This means that condition evaluation is something of an ongoing activity. If condition evaluation is viewed as an ongoing activity, then the possibility arises that preconditions and postconditions may be *re-evaluated* dynamically as the relevant state changes. Thus, a task that is provisionally

[1] In general, the question of when a task is ready to execute can be fairly complex, involving conditions on product and process state, scheduling constraints, resource availability, and so on. For the sake of simplicity, these are abstracted from the model shown here. These issues may be represented by additional states in the model (e.g., "scheduled" or "resources acquired"). Alternatively, the scope of preconditions may be considered to include all of the relevant issues.

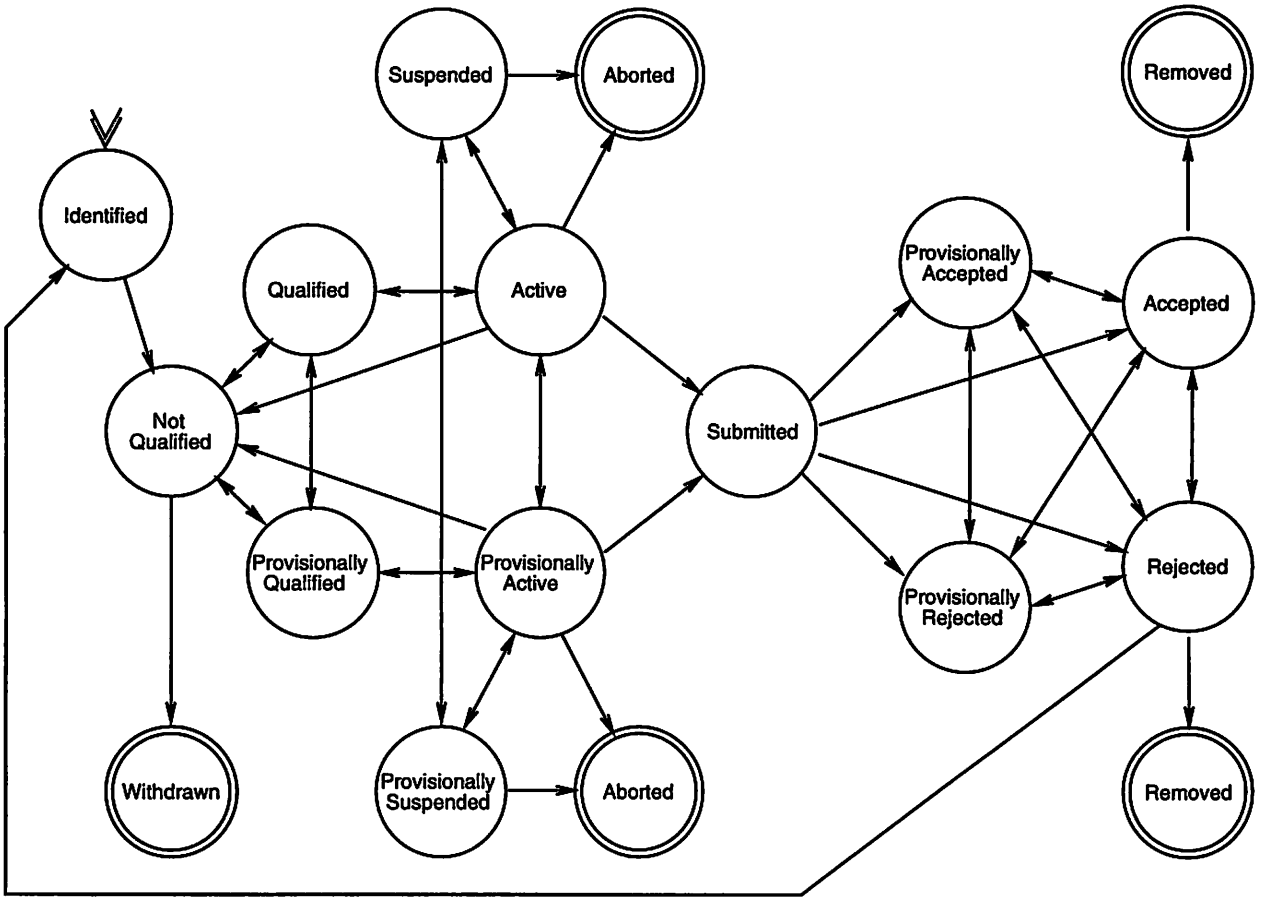

Figure 1: A State-Transition Model for Software-Process Tasks

qualified may become qualified if its indeterminate preconditions are found to be satisfied (that is certainly the hope). However, the inverse transition is also possible: a qualified task may become provisionally qualified if some previously satisfied precondition becomes indeterminate. Moreover, a task in either of those states may become not qualified if essential preconditions are found to be violated.

A task that is executing may be active or suspended. As with qualification, there are provisional and non-provisional versions of these states. Also, as preconditions may continue to be evaluated during task execution, transitions may be made between the active and provisionally active states, and the suspended and provisionally suspended states. This model admits the possibility that an active (or provisionally active) task may stop executing before it is complete (as a user may terminate an application before the associated job is finished); in that case, the task reverts to its respective qualified state. (For simplicity, a

suspended task is assumed to be activated before it is stopped.) As with the qualified states, it is also possible for an active or provisionally active task to revert to not qualified if a previously satisfied precondition becomes violated. An active or provisionally active task may also be aborted (although two aborted states are shown in the figure, these are logically equivalent and neither is provisional).

When an active task completes, it submits its results and terminates. The submitted state reflects this point in the task life-cycle. The submitted state marks the transition from the part of the life cycle where preconditions are relevant to the part of the life cycle where postconditions are relevant. As the entry of a task into the identified state signals that its preconditions are to be evaluated, so entry into the submitted state indicates that its postconditions are to be evaluated. From this point the task can only move forward in its life cycle. If there is a problem with the postconditions, the task is not re-activated; rather, it may be redone entirely.

An interesting issue is the possibility of submission from the provisionally active state. This may be considered illegal; in other words, a provisionally active task may be required to enter the (non-provisionally) active state before it can submit its results. Alternatively, if a task is able to complete its work in a provisionally active state, then there may be no reason *a priori* to proscribe it from submitting its results. It represents an important degree of freedom in the design and use of condition-based execution models.

Managerial control is relevant at several points in a task life-cycle such as that in Figure 1. Managerial discretion and direction may be exercised, for example, in determining when a provisional state may be entered (i.e., which conditions are not immediately required), when relevant conditions must be satisfied (e.g., when a task may move from the provisionally active state to the not qualified state), and whether a practically completed task may be submitted from the provisionally active state. Additionally managerial decisions may govern when a task is moved into any of the terminal states.

6 Discussion

This section addresses the implications of provisional execution for the concepts of preconditions and postconditions, the extent to which the need for provisional execution may be mitigated by more precise process representations, and generalizations of the basic provisional-execution model presented above.

6.1 Conceptual Issues

Provisional execution is intended to preserve conditional control of software processes according to preconditions and postconditions. However, it is also intended to allow for the managed introduction of a degree of flexibility that is not found in typical condition-based execution models. The introduction of this flexibility has the effect of stretching the usual notions of precondition and postcondition. A precondition need not be satisfied when a task starts; the condition may become satisfied only during the execution of the task. A postcondition need not be satisfied when a task finishes; the condition may only become satisfied some time later. In the extreme case, preconditions may not be satisfied at all, provided a task can satisfy its postconditions without them.

This stretching of traditional terminology suggests that some new or refined conceptual model is needed. At least a general distinction may be drawn between strict preconditions and provisional preconditions, and between strict postconditions and provisional postconditions. ("Strict" models would require the known satisfaction of all preconditions or postconditions.) A more precise terminology may be based on the times, relative to events in the execution life-cycle of a task, when a condition must be satisfied. These might include the times, for example, prior to initiation, at initiation, during execution, at termination, and subsequent to termination. Most current systems typically require or ensure that conditions are satisfied at initiation and termination. In some cases, though, a condition satisfied at initiation can be inferred to hold for at least some time into execution, and a condition satisfied at termination can be inferred to hold for at least some time subsequent to termination.

6.2 Appropriateness of Representation

It may be argued that a precondition is not really a precondition if it need not be satisfied at the time a task is invoked (and analogously for postconditions). Alternatively, if we did a better job of modeling our tasks, would we ever need preconditions or postconditions that could be temporarily ignored? For example, we might break up larger tasks into smaller subtasks to which preconditions and postconditions are more precisely matched. In this view, a provisional precondition on a higher-level task is a strict precondition on a subtask that executes some time into the higher-level task. A provisional postcondition on a subtask would actually be a strict postcondition on a higher-level task. Certainly precision of process representation is useful for many purposes, such as process analysis, understanding, planning, and measurement. However, three sorts of counterarguments suggest that provisional execution has a role to play even when processes are modeled with greater precision.

The first is that complete precision will not always be achievable. For example, it may not always be cost effective or technologically feasible to break tasks down into the finest possible subunits of conditional control. At that level of granularity, the process models may become too cumbersome or rigid (see, for example, [18]), or the costs of condition evaluation may become too burdensome. It may further be difficult to break down especially creative and contingent tasks into discreet subtasks and to anticipate the conditions that may affect their internal control.

The second counterargument is that some legitimate preconditions or postconditions may not be absolutely necessary. Preconditions and postconditions may represent and assure properties that make a variety of kinds of contribution to the execution of a task. Some properties may be truly necessary for functional correctness. Others may affect efficiency or cost, or help to improve product quality to some non-essential degree. For preconditions that are not truly critical, provisional execution may be justifiable under many circumstances.

The final counterargument, which is non-technical, is that the prospect of provisional execution is intuitively appealing. It allows for the exercise of engineering and managerial judgement, and provides for the "bending of rules" in response to the unanticipated contingencies and extra information that will affect even precisely defined processes. Although this argument is non-technical, it is not without technical implications and consequences. The main implication is that judgements must be made responsibly with respect to engineering constraints on the process. The main consequence is that opportunities for process and product improvement may be realized.

6.3 Generalizations

Provisional execution is proposed first of all as a means to accommodate indeterminacy in the evaluation of preconditions and postconditions. However, once the prospect of provisional execution is seen, the applicability of the model can be broadened. One generalization is to allow provisional execution not only when conditions are indeterminate but also when they are violated. This may be reasonable if the condition is expected to be satisfied soon or if it is deemed non-essential in a particular case. Another generalization is based on the database concept of compensating transactions as a technique for recovery from transaction failure [11]. The counterpart here is not to prevent or abort a task when a precondition or postcondition fails; rather, a compensating action may be invoked and the task allowed to proceed or stand.

7 Related Work

A number of the concepts that are bound up in this model of provisional execution are found in other contexts in computer science, including distributed computing, database transaction management, software analysis, and artificial intelligence.

Delayed Results The idea that the evaluation of a needed result may be delayed is embodied in *promises* in the Argus system [21]. A promise is a placeholder for a result that will be computed and returned via asynchronous remote procedure call. One of the advantages of promises, as for provisional execution, is that the computation of the value can proceed in parallel with the program that needs the value.

System-Application Conflicts The idea that conflicts may occur between application and system processes is also present in databases in the form of *extra-data* transactions [12]. In extra-data transactions, application transactions access some sort of data that are not among the application data managed by a database. These "extra" data may be outside the database system entirely (e.g., transient local variables) or part of the internals of the database-management system (e.g., transaction serialization graphs). A variety of concurrency and correctness issues result; the most relevant to provisional execution are when the transaction-management system and application transactions both access system data. This is somewhat analogous to situation when the process execution engine must evaluate preconditions or postconditions that refer to software product data. Here, data are shared again by the two categories of process, but the data lie in the application domain rather than the system domain. These two cases are illustrated in Figure 2.

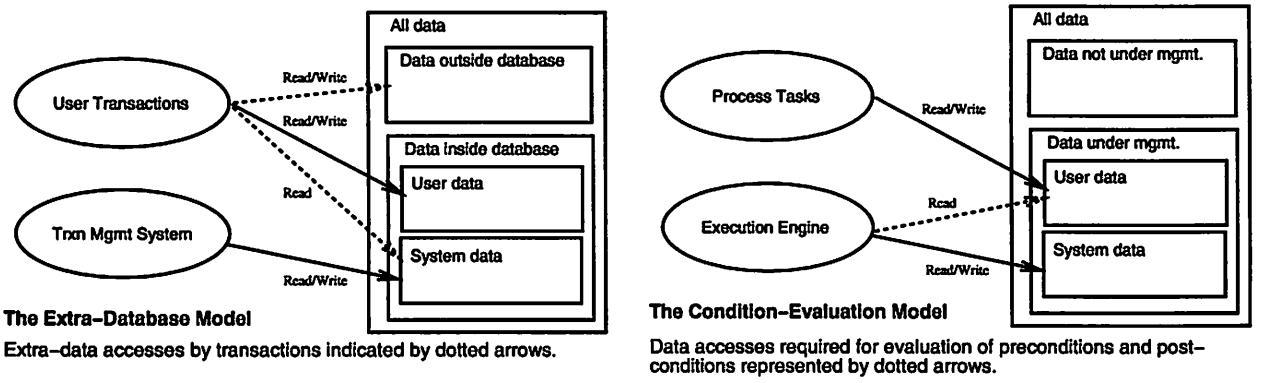

Figure 2: A Comparison of System-Application Concurrency Interactions

State Models of Task Execution State models have been widely used to represent process-task execution (e.g., [13]). The approach is perhaps epitomized by the ProcessWall [14], a language- and process-independent server for process state. Task states that are currently represented by the ProcessWall include *specified*, *initiated*, *scheduled*, *enabled*, *running*, *suspended*, *done*, and *aborted*. Provisional execution can be accommodated in this model at several points. The enabled state in particular signifies (among other things) that all preconditions are satisfied. It corresponds to the qualified state in Figure 1. As with the qualified, active, and suspended states in that figure, the enabled, running, and suspended states in the ProcessWall can be differentiated into provisional and non-provisional alternatives. The done state in ProcessWall signifies that all postconditions are satisfied. This state may also be differentiated into provisional and non-provisional alternatives.

Inconsistency and Incompleteness The idea that indeterminacy should be accommodated is related to the ideas of accommodating inconsistency and incompleteness. It has been argued that inconsistency of software data is unavoidable during software development and so mechanisms should be offered to allow work to proceed in the face of inconsistency [23, 1]. The corresponding point is made here that indeterminacy is unavoidable, although the reasons and response are different. The AdaPIC analysis tool-set [26] accommodates incompleteness of module specifications in the analysis of inter-module provide/request relationships. The incomplete parts have a formal representation and semantics, which are important in accommodating inconsistency and indeterminacy, as well.

Uncertainty Coping with uncertainty is a central problem in multi-agent systems, an emerging sub-discipline of artificial intelligence [20]. In these systems, individual agents must interact in the solution of interdependent subproblems. However, the interactions are generally imperfect due to limited communication bandwidth, heterogeneity among the agents, and dynamism in the environment. This is analogous to the situation in software processes when a task may be executed based on limited information regarding its preconditions or postconditions. The incorporation of provisional states into software-process execution models opens the door for the application of artificial intelligence techniques to accommodating indeterminacy in process control.

Optimism The idea that a task can proceed in the face of potential abort, which is essential to provisional execution, is also represented in optimistic concurrency-control for database transactions [9]. Optimistic concurrency control allows a transaction to run to completion and only then checks for possible concurrency conflicts with other transactions (at which point the completed transaction may be aborted). The protocol is optimistic be-

cause it presumes that usually there will be little interference between transactions and that transaction aborts will be relatively uncommon. Provisional execution does not require that process tasks run to completion before their viability is determined. However, it shares an optimistic outlook in that it presumes that conflicts will not usually stand in the way of the completion of the task. The transaction model defined in [16] makes use of both optimistic and pessimistic concurrency control for cooperation in software processes (where pessimistic concurrency control is based on locking).

Semantics-based Control The transaction model defined in [16] is also relevant to the case for provisional execution because it accepts that some activities may be allowed to run a greater risk of abort than others and that the process engineer or even process participant should be able to make judgements in this regard. This model can further be viewed as a high-level application of the ideas of semantic and programmable concurrency control [3], in which information about the semantics of activities is used to resolve conflicts between them. A semantic-based approach is equally applicable to the control of provisional execution, for example, in deciding which tasks may be executed provisionally, which conditions may be indeterminate, and when indeterminate conditions must finally be satisfied.

8 Summary

Condition-based execution models use preconditions and postconditions to control the the initiation and termination of tasks. These execution models are widely used in software processes. However, preconditions and postconditions to process tasks must generally depend on the same data on which process tasks operate. This gives rise to potential concurrency conflicts between the execution of process tasks and the evaluation of conditions that control the execution of those tasks. A number of strategies may be used to try to avoid these conflicts or mitigate their consequences; however, none of these strategies is without significant drawbacks. Provisional execution of software process tasks is intended to reduce the potential for concurrency conflicts in condition-based execution models while preserving high degree of concurrency. It does so at some risk to the execution of process tasks, but the risk is subject to management.

Provisional execution is based on the premises that the status of preconditions and postconditions may not always be known and thus that the execution models for process tasks should accommodate this indeterminacy. Consequently, the approach taken for provisional execution is to loosen the strict coupling between the evaluation of these conditions and the execution of associated tasks. More specifically, tasks may be provisionally initiated before

all of their preconditions are known to be satisfied, and provisionally terminated before all of their postconditions are known to be satisfied. However, the obligation to satisfy preconditions or postconditions remains. If these conditions are not satisfied by an appropriate point in the execution of the task or overall process then the task may be aborted or its results discarded. The use of provisional execution, the particular conditions that may be indeterminate, the points at which conditions must be satisfied, and other details in the control of provisionally executing tasks are all subject to programmed or human management.

The basic model of provisional execution suggests a number of corollaries, issues, and generalizations. Since preconditions and postconditions are evaluated asynchronously with respect to the execution of associated tasks, this opens the possibility that they may be reevaluated on an ongoing basis, with concomitant adjustments to task status. Since preconditions and postconditions may be evaluated over an extended period, it is necessary to determine just when, in the execution cycle of a task, preconditions and postconditions should apply (or, conversely, when they become irrelevant). It also admits the possibility that a provisional task may satisfy its postconditions before its preconditions are know to be satisfied. Although provisional execution is proposed as a way to accommodate indeterminacy in the evaluation of preconditions and postconditions, it may also be used to accommodate inconsistency. In other words, provisional execution may occur even when some preconditions and postconditions are false. Additionally, there are a variety of responses that may be made when an indeterminate precondition or postcondition is deemed to be ultimately not satisfied. For example, a dependent task maybe aborted; alternatively, the task may be allowed continue but with the addition of a compensating task.

Provisional execution shares concepts with a number of other areas in computer science, including optimistic and semantic concurrency control, state-based task execution models, asynchronous communication in distributed systems, and the accommodation of incompleteness, inconsistency, and uncertainty in software analysis, software product consistency maintenance, and the control of distributed agents.

Acknowledgements

This work was supported by the Advanced Research Projects Agency under Grant F30602-94-C-0137. Peri Tarr provided comments on an earlier report on the problem of concurrency conflicts in condition-based execution models (and the comments of anonymous reviewers of that report were likewise helpful). This work has also benefited from comments by the Arcadia process working group, especially Lee Osterweil and Dennis Heimbigner. Peri Tarr and Lori Clarke also pointed out work related to this report.

References

[1] Robert Balzer. Tolerating inconsistency. In *Proc. of the 13th International Conference on Software Engineering*, pages 158 – 165, May 1991.

[2] Sergio Bandinelli, Alfonso Fuggetta, and Sandro Grigolli. Process modeling in-the-large with SLANG. In *Proc. of the Second International Conference on the Software Process*, pages 75–83, 1993.

[3] N. Barghouti. *Concurrency Control in Rule-Based Software Development Environments.* PhD thesis, Columbia University, 1992.

[4] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269 – 317, September 1991.

[5] Noureddine Belkhatir, Jacky Estublier, and Melo L. Walcelio. ADELE-TEMPO: An environment to support process modeling and enaction. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 187 – 222. John Wiley & Sons Inc., 1994.

[6] Don Cohen. *AP5 Manual.* Univ. of Southern California, Information Sciences Institute, March 1988.

[7] R. Conradi, M. Hagaseth, J.-O. Larsen, M. N. Nguyên, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu. EPOS: Object-oriented cooperative process modelling. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33 – 70. John Wiley & Sons Inc., 1994.

[8] Wolfgang Deiters and Volker Gruhn. Managing software processes in the environment melmac. In *Proc. of the Fourth ACM SIGSOFT Symposium on Practical Software Development Environments*, pages 193–205, 1990. Irvine, California.

[9] Ahmed K. Elmagarmid and Y. Leu. An optimistic concurrency control algorithm for heterogeneous distributed databse systems. *IEEE Data Engineering*, pages 26 – 32, 1987.

[10] Christer Fernström. PROCESS WEAVER: Adding process support to UNIX. In *Proc. of the Second International Conference on the Software Process*, pages 12 – 26, 1993.

[11] Hector Garcia-Molina and Kenneth Salem. Sagas. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of the 1987 SIGMOD International Conference on the Management of Data*, pages 249–259. ACM, May 1987.

[12] Narain Gehani, Krithi Ramamritham, and Oded Shmueli. Accessing extra database information: Concurrency control and correctness. Technical Report CMPSCI Technical Report 93-81, Computer Science Department, University of Massachusetts, Amherst, Massachusetts, 01003, August 1993.

[13] Dennis Heimbigner. The process modeling example problem and its solutions. In *Proc. of the First International Conference on the Software Process*, page 174, 1991. Redondo Beach, California, October, 1991.

[14] Dennis Heimbigner. The ProcessWall: A Process State Server Approach to Process Programming. In *Proc. Fifth ACM SIGSOFT/SIGPLAN Symposium on Software Development Environments*, Washington, D.C., 9-11 December 1992.

[15] Karen E. Huff and Victor Lesser. A plan-based intelligent assistent that supports the software development process. In *ACM Symposium on Practical Software Development Environments*, pages 97 – 106, 1988.

[16] G. Junkermann, B. Peuschel, W. Schäfer, and S Wolf. MERLIN: Supporting cooperation in software development through a knowledge-based environment. In Anthony Finkelstein, Jeff Kramer, and Bashar Nuseibeh, editors, *Software Process Modelling and Technology*, pages 103 – 129. John Wiley & Sons Inc., 1994.

[17] Gail E. Kaiser, Naser S. Barghouti, and Michael H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In Bruce Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, January 1990.

[18] Takuya Katayama and Sumio Motizuki. What has been learned from applying a formal process model to a real process. In *Proc. 7th International Software Process Workshop*, 1991. Yountville, California.

[19] Programming Systems Laboratory. Marvel 3.0 administrator's manual. Technical Report TR CUCS-032-91, Columbia University, New York, New York 10027, 1991.

[20] Victor R. Lesser. Multiagent systems: An emerging subdiscipline of AI. *ACM Computing Surveys*, 1995. to appear.

[21] Barbara Liskov and Liuba Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. Technical Report Programming Methodology Group Memo 60-1, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, March 1988.

[22] Stanley M. Sutton, Jr. FCM: A flexible consistency model for software processes. Technical Report CU-CS-462-90, University of Colorado, Department of Computer Science, Boulder, Colorado 80309, 1990.

[23] Stanley M. Sutton, Jr. A flexible consistency model for persistent data in software-process programming languages. In Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors, *Implementing Persistent Object Bases – Principles and Practice*, pages 305–318. Morgan Kaufman, 1991.

[24] Stanley M. Sutton, Jr., Dennis Heimbigner, and Leon J. Osterweil. APPL/A: A language for software-process programming. *ACM Trans. on Software Engineering and Methodology*, 4(3), July 1995. to appear.

[25] Peri L. Tarr and Lori A. Clarke. PLEIADES: An Object Management System for Software Engineering Environments. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 56–70, December 1993.

[26] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. The adapic toolset: Supporting interface control and analysis throughout the software development process. *IEEE Trans. on Software Engineering*, 15(3):250–263, March 1989.