To Appear in *Proc. of the 8th European Workshop on Software Process Technology.*19-21 June 2001. Witten, Germany.

Applying Real-Time Scheduling Techniques to Software Processes: A Position Paper

Aaron G. Cass and Leon J. Osterweil

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
{acass, ljo}@cs.umass.edu

Abstract. Process and workflow technology have traditionally not allowed for the specification of, nor run-time enforcement of, real-time requirements, despite the fact that time-to-market and other real-time constraints are more stringent than ever. Without specification of timing constraints, process designers cannot effectively reason about real-time constraints on process programs and the efficacy of their process programs in satisfying those constraints. Furthermore, without executable semantics for those timing specifications, such reasoning might not be applicable to the process as actually executed. We seek to support reasoning about the real-time requirements of software processes. In this paper, we describe work in which we have added real-time specifications to a process programming language, and in which we have added deadline timers and task scheduling to enforce the real-time requirements of processes.

1 Introduction

Software process research is based on the notion, borrowed from manufacturing, that the way to build a good product is to have a good process by which that product is developed. Process programming languages were developed to help describe and make more repeatable the good processes of software development. These processes mostly involve human agents that do the bulk of the work. However, at specified points in the process, an automated agent might be introduced to perform some portion of the task, such as compiling the source code or executing a test. Software development processes, consisting of both automated and human-executed tasks, have real-time requirements. Phases of development must be done in timely fashion in order to allow enough time for later phases before the ship date arrives. At a finer-grained level, automated tasks can and should be done quickly to avoid delaying the overall progress of the development effort and to keep the human engineers from needlessly waiting.

In this work, we have extended an existing process programming language to provide deadline timers and dynamic scheduling to ensure that tasks will be completed within specified timing constraints.

As part of our ongoing research in process programming, we have developed Little-JIL [9, 10], a hierarchical, graphical, executable, and semantically rich process programming language. We are interested in applying scheduling techniques to process programs written in Little-JIL. As an example, consider a software design process. Clearly, a software design for a large software system requires human efforts on a scale of many designers working together for days, weeks, and months. However, there are points in the process where tasks are carried out that need to be done automatically and quickly, and as software engineering becomes better understood, the percentage of the process that can be done automatically and should be done quickly will increase. One area that can and should be automated is the periodic checking of the conformance of the design with respect to various design rules. If the conformance checking is not done in a timely fashion, the next design iteration will be delayed, the fixing of conformance errors will be delayed, and the project with either lack desired features or fail to meet time-to-market goals. This tension between tasks with loose, long deadlines and tasks with tight, short deadlines represents the key real-time characteristic of the kind of tasks we are trying to solve in this work.

A portion of a software design process program, including design conformance checking, is shown in Figure 1. The Add Component step at the root of the tree is a sequential step (indicated by the arrow), which means that the substeps must execute in left to right order without overlap. Therefore, the designer must first define a component and then integrate it in the overall design. After that, design conformance checking is performed, in this case by automated checkers.

We seek an approach that will allow the flexible specification of real-time requirements in a variety of scenarios. One scenario of interest is one in which overall design phase timing constraints are specified in addition to the real-time requirements of conformance checking. Another common scenario my be one in which we can't precisely estimate the time it takes to define and integrate a component, but yet we can give fairly accurate and precise estimates of the conformance checking activities because they are performed by automated checkers. Furthermore, these estimates will improve on each iteration through the recorded experience of past checks.

We therefore seek an approach that allows the enforcement of real-time schedules for all or part of a process program. In the rest of this paper, we further study the scenario in which the conformance checking sub-process is the only part of the process that has real-time requirements.

In this conformance checking sub-process, we would like to get as much useful conformance checking done as is possible in the time given. There are different categories of checks which can be done in parallel (as indicated by the two horizontal lines in the Perform All Checks step), each with current estimates of how long they will take to perform (perhaps based on previously recorded experience). If possible, we would like to perform all checks as the process specifies. However, there are real-time failures that might preclude this. In this work, we aim to deal with at least two kinds of real-time failure:

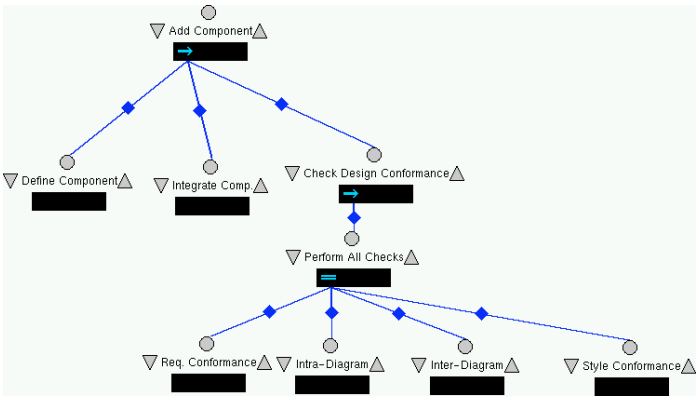

Fig. 1. An example Little-JIL process program

- Scheduling failure – If we can determine, based on current estimates of how much time each task will take, that it is impossible to complete all the conformance checks in the alloted time, then we must choose a different set of checks to perform.
- Deadline expiration – Once scheduled, tasks might overrun their deadlines if the timing estimates were inaccurate, or if the agents involved are busy with other tasks. In this case, we need to choose a different set of checks to perform, based on how many we have performed and how much time we have left to perform checks.

1.1 Differences from traditional real-time problem domain

The kinds of problems that can be expressed in Little-JIL are different from the traditional real-time problems in many ways. Primary among these is the fact that some tasks are done by humans, while others are done by software tools and therefore some tasks have tighter deadlines than others. Many tasks might take hours, days, or weeks while others expect a response time of seconds or less. In fact, because of the looseness of some of the deadlines, it may be desirable to not even include some tasks in a real-time schedule, choosing only to schedule those tasks that have tight deadlines. We seek a flexible approach that allows the mixing of coarse timing constraints with fine-grained ones. We seek an approach that allows for not scheduling some tasks (perhaps human-executed tasks) with coarse timing constraints while still scheduling the time-critical tasks.

2 Related Work

Other workflow and process languages have included timing notions. For example, the workflow language WIDE [2, 3] allows the specification of a delayed start to a task, which can be used to check to see if another activity was accomplished

during the delay. Their approach integrates the real-time failure handling with the normal control flow instead of separating this exceptional behavior.

There is a similar mechanism in SLANG [1]. A timeout duration can be specified between two transitions, called the start transition and the end transition, in the Petri-Net. If the end transition is reached, the current time is checked to ensure that it is within the specified duration after the firing of the start transition. If it is not, the end transition's TO (timeout) half is fired. This mechanism requires that transitions must be modified in order to be allowable end transitions for the timeout duration – thus the specification of timeouts is not separated from the specification of normal flow and task decomposition.

Our work is also related to work in real-time scheduling. For example, Liestman and Campbell [6] introduce primary and alternate algorithms for computing solutions of differing quality and timing characteristics. They then attempt to schedule the primary alternative, but if no valid schedule can be constructed, they attempt the alternate algorithm. Liu et al. [7] provide dynamic approaches to scheduling calculations that are logically broken into mandatory and optional subtasks.

3 Our Approach

Our approach has been two-fold. We first provide a simple deadline timer mechanism for the specification of maximum durations of leaf steps and then we use this information to produce schedules of complex structures.

3.1 Deadline Timers

Our approach has been to allow the specification of maximum durations on step specifications in Little-JIL processes, as shown in Figure 2, in which the leaf conformance checking steps are assigned durations of 150 time units. At run-time, the duration, if specified, is used to initialize a timer. Then, if the agent for the step executes the step within the duration, the step completes normally. However, if the timer expires before the step is done, the Little-JIL run-time system [4] causes a Little-JIL exception to be thrown.

This exception is handled in the same way as other exceptions are handled according to the language semantics [9]. The exception mechanism takes advantage of the hierarchical structure of Little-JIL programs to provide natural scopes for exception handling. When a step throws an exception, the parent step has the first opportunity to react to the exception. It can do so by executing a handler, which is a potentially complex Little-JIL step, or it can throw the exception further up the hierarchy. In this way, the exception can be handled at the scope at which it is most naturally handled.

Because we have treated deadline expiration as an exception like any other, we can provide some real-time assurances with relatively little additional language or run-time complexity. As an example, consider the Little-JIL program shown in Figure 2, an extension of the one shown in Figure 1. In this program,

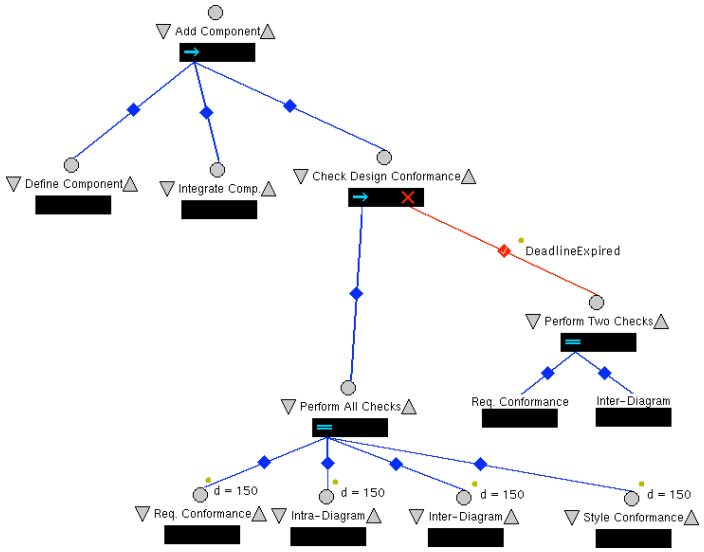

Fig. 2. A Little-JIL process program with a deadline expiration exception handler.

the DeadlineExpired exception is handled by performing the two most important checks instead of all four that are specified in the normal process description.

3.2 Real-Time Scheduling

As we have detailed in [5], we have integrated real-time scheduling techniques into Little-JIL. We have adapted a real-time heuristic-search based dynamic scheduling algorithm [8] to work with the hierarchical structures of Little-JIL programs. The scheduler we have developed takes both timing estimates and resource requirements into account to determine when steps can be executed. Where there is no resource contention, the scheduler allows two steps to be scheduled to execute in parallel. Once this scheduler was adapted to work with Little-JIL structures, we updated the Little-JIL run-time to call the scheduler at appropriate times. When the Little-JIL run-time attempts to execute a step, if that step has a timing specification, the scheduler is called to determine if the estimates of the running-times of the substeps are consistent with the timing specification of this newly executing root step.

The scheduler uses a heuristic-search approach to arrive at a schedule, a set of starting times for the steps in a Little-JIL process. If no such schedule is possible, the scheduler can cause a SchedulingFailure exception to be thrown. Just as any other Little-JIL exception, this exception can be handled at any higher scope. We have shown an example process in Figure 3 that shows the use of a handler for this type of exception to perform a different set of conformance checks. This is much like the deadline expiration example except that in the case

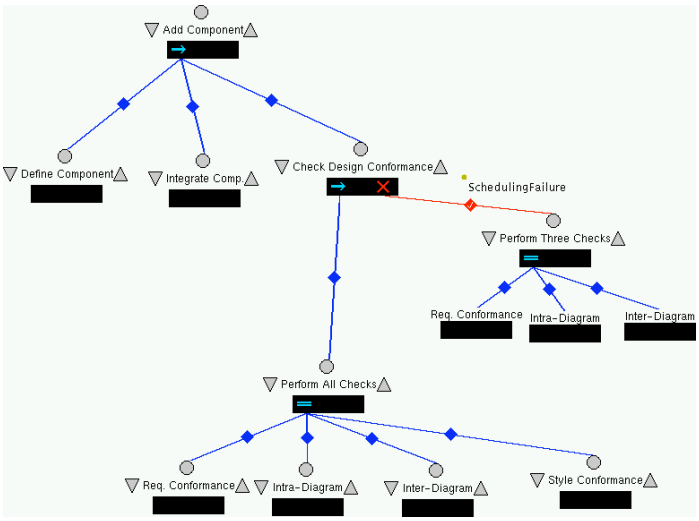

Fig. 3. A Little-JIL process program with a scheduling failure exception handler.

of scheduling failure, the exception occurs before the substeps are executed and so there is more time to execute an effective fix.

4 Conclusion

Because real-world software development processes have timing requirements about which process designers need to reason and provide assurances, we have added the ability to specify and enforce real-time constraints in Little-JIL. Our approach provides flexibility in the specification of real-time requirements, and allows for the handling of two different kinds of real-time failures. The approach uses the exception handling mechanisms of Little-JIL to keep the specification of the handling of real-time failures separate from the specification of normal process flow.

Acknowledgements

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032 and by the U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the Defense Advanced Research

Projects Agency, the Air Force Research Laboratory/IFTD, the U.S. Dept. of Defense, the U. S. Army, or the U.S. Government.

References

1. S. Bandinelli, A. Fuggetta, and S. Grigolli. Process modeling in-the-large with SLANG. In *Proc. of the Second Int. Conf. on the Soft. Process*, pages 75–83. IEEE Computer Society Press, 1993.
2. L. Baresi, F. Casati, S. Castano, M. G. Fugini, I. Mirbel, and B. Pernici. WIDE workflow development methodology. In *Proc. of the Int. Joint Conf. on Work Activities, Coordination, and Collaboration*, 1999. San Francisco, CA.
3. F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Trans. on Database Systems*, 24(3):405–451, Sept. 1999.
4. A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and A. Wise. Logically central, physically distributed control in a process runtime environment. Technical Report 99-65, U. of Massachusetts, Dept. of Comp. Sci., Nov. 1999.
5. A. G. Cass, K. Ramamritham, and L. J. Osterweil. Exploiting hierarchy for planning and scheduling. Technical Report 2000-064, U. of Massachusetts, Dept. of Comp. Sci., Dec. 2000.
6. A. L. Liestman and R. H. Campbell. A fault-tolerant scheduling problem. *IEEE Trans. on Soft. Eng.*, 12(11):1089–95, Nov. 1986.
7. J. W. Liu, K. Lin, W. Shih, A. C. Yu, J. Chung, and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer*, 24(5):58–68, May 1991.
8. K. Ramamritham, J. A. Stankovic, and P. Shiah. Efficient scheduling algorithms for real-time multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):184–194, Apr. 1990.
9. A. Wise. Little-JIL 1.0 Language Report. Technical Report 98-24, U. of Massachusetts, Dept. of Comp. Sci., Apr. 1998.
10. A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton, Jr. Using Little-JIL to coordinate agents in software engineering. In *Proc. of the Automated Software Engineering Conf.*, Sept. 2000. Grenoble, France.