2000

Verifying Properties of Process Definitions

Jamieson M. Cobleigh
University of Massachusetts Amherst

Verifying Properties of Process Definitions

Jamieson M. Cobleigh, Lori A. Clarke, Leon J. Osterweil
Department of Computer Science
University of Massachusetts at Amherst
Amherst, MA 01003-6410
[jcobleig, clarke, ljo]@cs.umass.edu

ABSTRACT

It seems important that the complex processes that synergize humans and computers to solve widening classes of societal problems be subjected to rigorous analysis. One approach is to use a process definition language to specify these processes and to then use analysis techniques to evaluate these definitions for important correctness properties. Because humans demand flexibility in their participation in complex processes, process definition languages must incorporate complicated control structures, such as various concurrency, choice, reactive control, and exception mechanisms. The underlying complexity of these control abstractions, however, often confounds the users' intuitions as well as complicates any analysis.

Thus, the control abstraction complexity in process definition languages presents analysis challenges beyond those posed by traditional programming languages. This paper explores some of the difficulties of analyzing process definitions. We explore issues arising when applying the FLAVERS finite state verification system to processes written in the Little-JIL process definition language and illustrate these issues using a realistic auction example. Although we employ a particular process definition language and analysis technique, our results seem more generally applicable.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software process*

1. INTRODUCTION

Processes are pervasive in areas of human and computer interaction [5, 8, 9]. At the same time, societal vulnerability to poor quality in these processes has become worrisome. Thus, we advocate rigorous analysis of process definitions to demonstrate that they are free from faults that could lead to serious failures.

In earlier work it has been suggested that processes are a particular kind of software [13] and that they should be developed, verified, and evolved using approaches analogous to those used for application software. In earlier work it has also been suggested that static analysis approaches are effective in helping to reason about software systems. In this paper, we demonstrate the applicability and benefits, as well as some of the research challenges, associated with applying finite state verification to process definitions. Although software processes are software too, process software has characteristics that tend to differentiate it from most conventional application software in ways that complicate static analysis. Thus, while the need for analysis remains strong, the complications in doing so are noteworthy.

Ongoing process research suggests that graphical process models are useful in raising human awareness and intuition about process characteristics. Unsurprisingly, the most effective models incorporate high-level abstractions that support concise visualization. While intuitively appealing, we have found that such process models often entail subtleties that lead to error-prone process definitions. We have developed a visual language, Little-JIL, that provides a range of process abstractions that have proven to be effective for describing human and computer interaction. In this paper we describe how FLAVERS [4], a finite state verification system, has been used to verify properties of processes that have been defined using Little-JIL. The paper demonstrates that process abstractions can be quite effective in supporting precise process definitions, but the underlying semantic complexity poses challenges for static analysis. Our work addresses those challenges, and, in doing so, provides experience that should be of importance for future research in both finite state verification and in process language design. Although we present an example in terms of a particular process definition language and a particular analysis technique, we contend that the insights gained are also applicable to other process definition languages [e.g., 1, 5, 6, 11] and other static analysis techniques [e.g., 3, 7, 10].

This work expands on the types of analyses that have been explored for process definition systems. Several process languages, including IDEF0 [11], ProcessWeaver [5], and Statemate [6], allow limited types of static analysis, such as type checking and other consistency checking. Perhaps the most ambitious static analysis is carried out in the FunsoftNets system [1]. This system uses a Petri Net-like model to define processes. The system incorporates analyzers that evaluate well-formedness and detect such defects as deadlocks and traps in the underlying Petri Net.

2. LITTLE-JIL

Little-JIL is an expressive process definition language that uses a graphical notation that helps users quickly grasp the meanings of process definitions [14]. Here we present a subset of Little-JIL,

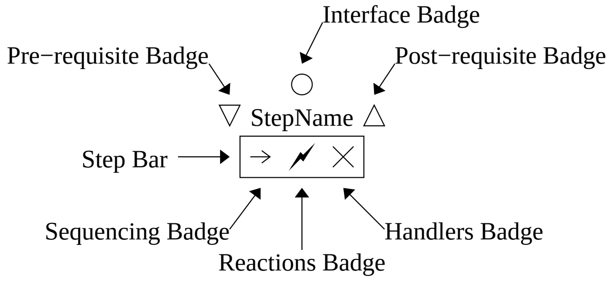


Figure 1: A Little-JIL step

so that the motivating example can be understood. Little-JIL has well defined formal semantics that allow Little-JIL definitions to be executed and analyzed. A Little-JIL process definition describes the coordination of activities of agents, where an *agent* is an entity, either human or computer, that can be assigned work to do. In Little-JIL, *steps* represent work that can be assigned to an agent.

Steps: Each step in a Little-JIL definition is represented by a step icon as shown in Figure 1. Each step is given a name and has a set of badges that represent key information about the step, including the step's control flow, the exceptions the step handles, the parameters needed by the step, and the resources needed to execute the step. Each step can only be declared once in a Little-JIL definition, but a step can be referenced many times in the process definition. These additional references are depicted by a step with its name in italics and no badges.

Step Execution: The execution semantics of a Little-JIL step are defined by a finite state machine, whose behavior can be summarized by five states: posted, retracted, started, completed, and terminated. A step is moved into the posted state when it is eligible to be started. A step is moved into the started state when the step's agent begins executing the step. When the work specified by a step is successfully finished, the step moves to the completed state. If the step cannot be successfully completed, it moves to the terminated state. A step is put into the retracted state if it had been posted, but not started, and is no longer eligible to be started. The step's execution can end when it is in the retracted, completed, or terminated state.

Sequencing Badges: A Little-JIL process is represented by a tree structure where children of a step are the substeps that need to be done to complete that step. The parent-child relation is depicted by a line between the child and the parent's sequencing badge. All non-leaf steps must have a sequencing badge, which describes the order in which its substeps are performed. The four different supported sequencing types are shown in the key in Figure 2

A *sequential step* indicates that its substeps are to be performed one at a time, from left to right. A *parallel step* indicates that its substeps can be done concurrently, and that the step is completed if and only if all of its substeps have completed. A *choice step* indicates that a step's agent must make a choice among any of its substeps. All of the substeps are available to be performed, but only one can be selected at a time. If a selected substep completes, then the choice step completes. A *try step* attempts to perform its substeps in order, from left to right, until one of them completes. If a substep terminates, then the next substep is tried.

Exception Handling: There is considerable evidence that processes have complex exception structure. Thus, steps in Little-JIL can throw exceptions, which are caught by the nearest ancestor having a matching handler, as indicated by the ancestor's handler badge. To concisely represent complex exception handling, Little-JIL enables handlers to be steps, so they may have a full hierarchical structure. Our experience also indicates that articulate expression of process exceptional flow is facilitated by the attachment of any of four different kinds of handler control-flow badges that indicates how the step catching the exception should proceed after the handler completes. These are shown in the key of Figure 2.

When a handler with a *restart* badge completes, the step catching the exception is restarted. When a handler with a *continue* badge completes, the step catching the exception continues as if the substep that generated the exception completed normally. When a handler with a *complete* badge completes, the step catching the exception moves into the completed state. When a handler with a *rethrow* badge completes, the step catching the exception terminates and rethrows the exception. Some handlers consist only of a badge, but no step structure.

Requisites: Process definitions seem to benefit substantially from the attachment of pre- and post-requisites to steps. These constructs are natural vehicles for monitoring agent performance of steps and support the retention of process control, while still granting the agent latitude and initiative in step execution. Thus, a step in Little-JIL can have pre- and post-requisites. A *pre-requisite* is performed after a step starts, but before the work of the step can be initiated. A *post-requisite* has to be done before a step can complete. A failure of a requisite for a step throws an exception that is handled by the matching handler at the step's nearest ancestor. This failure terminates the step with the requisite.

Interface Badges: Artifact flow and resource specification have both been found to be essential to the precise definition of realistic processes. In Little-JIL, interface badges are used to declare what parameters a step has, what exceptions it throws, and what resources it needs. Parameters declared in a Little-JIL step have a name, type, and mode. The name is used to identify the parameter and the type declares what type of object the parameter is. Little-JIL uses copy-in/copy-out semantics for parameter passing and a parameter may have one of four modes. An *in parameter*, denoted by a down arrow, is passed from the parent and its value should be copied when the step starts. An *out parameter*, denoted by an up arrow, is passed to the parent, which must copy the value when the step completes. An *in-out parameter*, denoted by an up-down arrow, indicates the value of the parameter should be copied in when the step starts and copied out when the step completes. A *local parameter*, indicated by a diamond, is created by a step to allow passing of parameters between that step and its descendants.

3. MOTIVATING EXAMPLE

The utility of these constructs can perhaps be seen best through an example. Thus, this section demonstrates the use of Little-JIL to define an auction, a process that is gaining increasing prevalence in ecommerce. In an auction process, a buyer and seller reach an agreement about an acceptable price for an item. The process is supervised and controlled by a third party, the auctioneer. One type of auction is the Open-Cry Auction. In its most common form, bidding starts at a low price and the price is increased as bidders offer successively higher prices. The auction closes when one bidder has offered a price that is higher than what any other bidder is willing

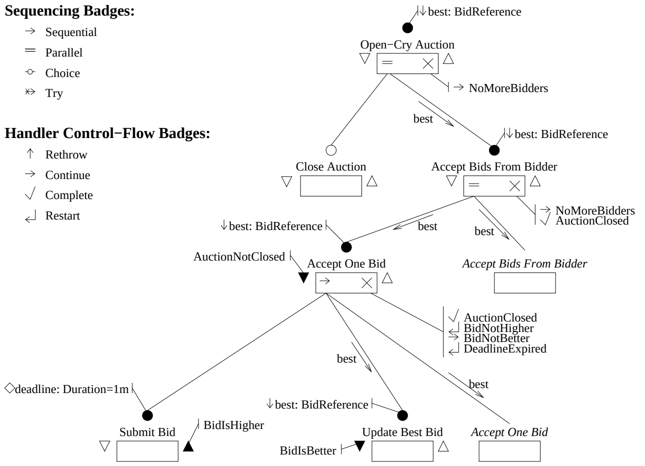


Figure 2: Open-Cry Auction Process

to offer within some time frame. The high bidder is then awarded the item and has to pay the amount of their highest bid. With online auctions the auctioneer is not a person but a program and the bidders are distributed across a network. At present bidders are usually humans, but it is expected that bidding will increasingly be carried out by automated agents. Thus, auctions will be carried out in a more rapid fashion, with decreasing amounts of human interaction and scrutiny. For these reasons, having some way to ensure that the activities of the auctioneer and bidders proceed in expected ways is important.

Many different properties of an Open-Cry Auction should be verified. For example, no bids should be accepted after an auction has been closed. It is also important to verify that the auctioneer considers all bids that are submitted and that the person submitting the highest bid is actually awarded the item and at the highest bid price. If parts of the auctioneer or bidder are carried out by a computer, then it is important to check to ensure that the computer software cannot deadlock and does not have any undesirable race conditions.

While it may be possible to verify these properties by the direct analysis of the code used to implement such an auction, the analysis of a higher level representation generally offers advantages of scalability and early fault detection. A Little-JIL definition is just such a higher level representation. Figure 2 shows a simplified version of an Open-Cry Auction written in Little-JIL [2].

At a high level, this process is very straightforward. In this process, performing an auction is broken down into two steps that can happen in parallel, "Close Auction" and "Accept Bids From Bidder". One "Accept Bids From Bidder" step is created for each bidder in the auction. Each bidder is handed off to an "Accept One Bid" step, which is responsible for handling a single bidder's bids. These "Accept One Bid" steps can happen in parallel, so multiple bidders can be submitting bids simultaneously. The process of accepting one bid is done by having the bidder submit a bid, and then having the auctioneer update the best bid depending upon whether or not the bid just submitted is higher than the current high bid. This process recurses on "Accept One Bid" so that each bidder can continue to submit bids until the auction is closed.

4. FLAVERS

FLAVERS (**FL**ow **A**nalysis for **VER**ification of **S**ystems) is a static analysis tool that can verify user specified properties of sequential and concurrent systems [4]. Like all automated verification systems, FLAVERS requires an accurate model of the computation upon which to base the analysis. The model FLAVERS uses is based on annotated *Control Flow Graphs* (CFG). Annotations are placed on nodes of the CFGs to represent events that occur during execution of the actions associated with a node. Since a CFG corresponds to the control flow of a sequential system, this representation is not sufficient for modeling a concurrent system. FLAVERS uses a *Trace Flow Graph* (TFG) to represent concurrent systems. The TFG consists of a collection of CFGs with *May Immediately Precede* (MIP) edges between tasks to show intertask control flow. A CFG, and thus a TFG, over-approximates the sequences of events that can occur when executing a system.

FLAVERS requires that a property to be checked be represented as a Finite State Automaton (FSA). FLAVERS uses an efficient state propagation algorithm to determine whether all potential behaviors of the system being analyzed are consistent with the property. FLAVERS will either return *conclusive*, meaning the property being checked holds for all possible paths through the TFG, or *inconclusive*, meaning FLAVERS found some path through the TFG that causes the property to be violated. FLAVERS analyses are conservative, meaning FLAVERS will only return conclusive results when the property holds for all TFG paths. If FLAVERS returns inconclusive results, this can either be because there is an execution that actually violates the property or because the property is violated on infeasible paths through the TFG. *Infeasible paths* do not correspond to any possible execution of the system but are an artifact of the imprecision of the model. If the inconclusive result is because of infeasible paths, then the analyst can introduce *feasibility constraints*, which are also represented as FSAs, to improve the precision of the model and thereby eliminate some infeasible paths from consideration. An analyst might need to iteratively add feasibility constraints and observe the analysis results several times before determining whether a property is conclusive or not. Feasibility constraints give analysts some control over the analysis process by letting them determine exactly what parts of a system need to be modeled in order to prove a property.

FLAVERS' state propagation has worst-case complexity that is $\mathcal{O}\left(N^2 \cdot |S|\right)$, where N is the number of nodes in the TFG, and $|S|$ is the product of the number of states in the property and all constraints. In our experience, a large class of important properties can be proved by using only a small set of feasibility constraints.

5. MODELING PROCESSES

Earlier we described the sequencing badges supported by Little-JIL. For each of these step kinds, we constructed a CFG model. In Little-JIL, the types of exception handlers on a step can affect the model. Space does not permit us to provide all models of all possible combinations of steps and exception handlers here. Rather, we illustrate the models of each step kind using one kind of exception handler, usually one that simplifies the model for that step.

Leaf Steps: The model for a Leaf Step is shown in Figure 3. Control flows in from the parent of the Leaf Step and the step is posted. After being posted, the step can be started. From the started state, the step can either complete or terminate. A pre-requisite can be added by putting its model between the "LeafPosted" and "LeafStarted" node. A post-requisite can be added by putting its model

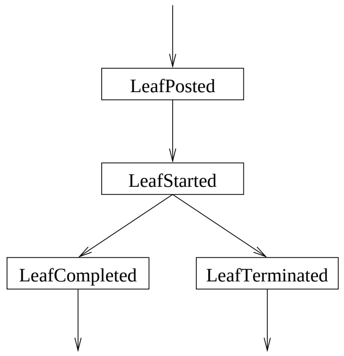


Figure 3: Model of a Leaf Step

immediately before the "LeafCompleted" node. Since steps terminate if their requisites terminate, the model should have the terminated path out of the pre- and post-requisites connected to the "LeafTerminated" node.

Sequential Steps: A sequential step performs the work of all of its substeps, one at a time, from left to right. Suppose, for simplicity, the sequential step has rethrow handlers for any exception thrown by its substeps. This means that when an exception is thrown, the sequential step terminates. This model, generalized to n substeps, is shown in Figure 4. As before, flow comes in from the sequential step's parent and it is posted and then started. At this point, the sequential step attempts to do its first substep. This is a recursive model, so the model for the first substep is represented by the oval labeled Substep$_1$. If Substep$_1$ completes, the process moves on to the next substep and continues in this fashion until Substep$_n$ is reached. If Substep$_n$ completes, the sequential step completes. If any substep terminates, then the sequential step terminates.

Parallel Steps: A parallel step allows the work of its substeps to proceed concurrently. As with the sequential step, we assume for simplicity that the parallel step has only rethrow handlers. While the parallel step may in general have n substeps, for simplicity we show a parallel step that has only two substeps. The model of this step, as shown in Figure 5, has a dashed edge in it to represent interactions that may occur due to concurrency. In particular, the dashed edge represents a set of FLAVERS MIP edges, which are used to represent the ways in which flow can move between different tasks. The dashed edge in this figure represents the addition of MIP edges between every pair of nodes in Substep$_1$ and Substep$_2$.

In addition, the parallel step cannot finish until all of its substeps have finished. The potential parallelism involved makes representing this behavior directly in a TFG difficult, so we have chosen to use FLAVERS' feasibility constraint mechanism to ensure that the parallel step cannot complete or terminate until all of its substeps have finished. This approach is consistent with how FLAVERS models some of the concurrency constructs in Java [12].

Try Steps: Although our Open-Cry Auction process example does not include any choice steps or try steps, we briefly describe how their semantics can be modeled. Try steps are designed to try their substeps one at a time, in order, until one completes. For the model shown in Figure 5, we assume a try step has only continue exception handlers, so that the try step can attempt all of its substeps. The try step begins by attempting Substep$_1$. If an attempted substep completes, the step completes, but if it terminates, the process moves on to the next substep. If any of the substeps completes, the try step completes; if all of the substeps terminate, the try step terminates.

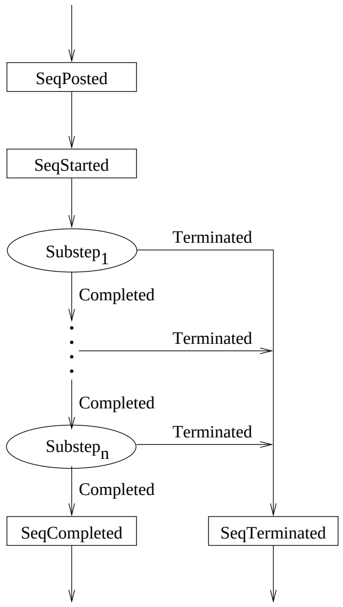
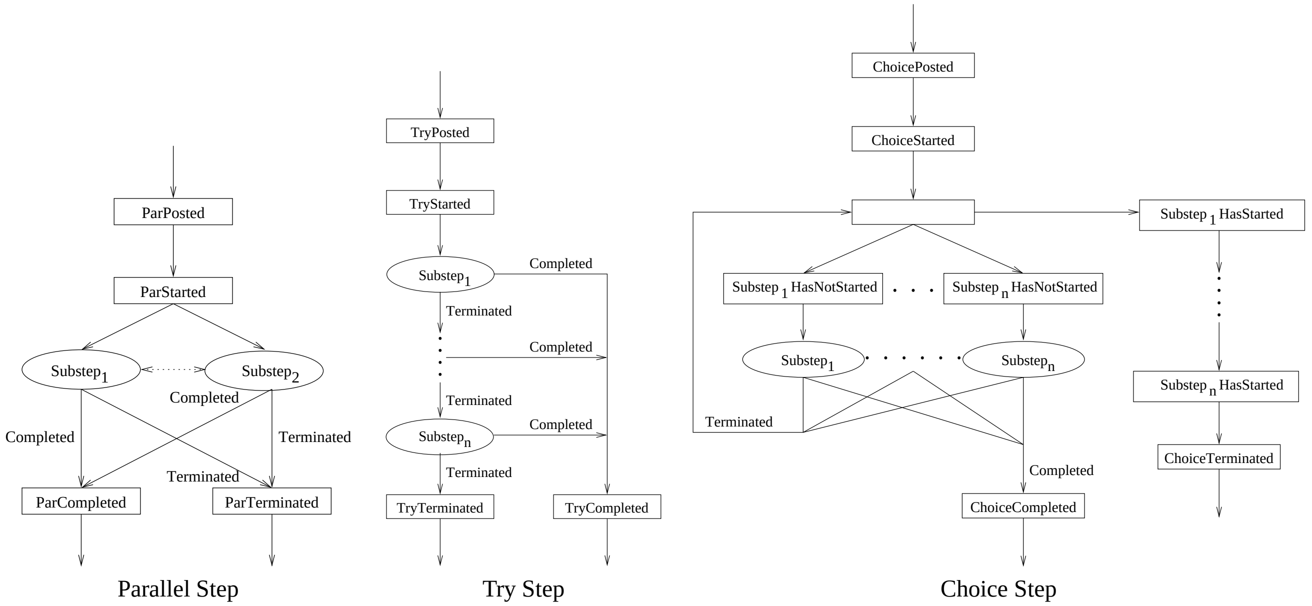


Figure 4: Model of a Sequential Step

Choice Steps: If there are n substeps to a choice step, it is possible that all n substeps might be tried before the choice step finishes. There are 2^n subsets of the n substeps. For an analysis to be conservative, it may need to consider not only all of these subsets, but all orderings of the substeps within each subset. Even though this can present a challenge for analysis, experience has shown that human agents desire the empowerment that such constructs provide.

Figure 5 shows the CFG for the Choice step. The astute reader will notice that this model contains many infeasible paths, since there is nothing to prevent FLAVERS from considering paths where a substep is started several times. We use a set of feasibility constraints, each similar to the one shown in Figure 6, to restrict the paths that will be traversed during analysis. These constraints prevent FLAVERS from starting substeps more than once and from terminating the Choice Step before all of the substeps have been attempted. We chose to use this model because representing all of the orderings of the substeps in the CFG explicitly would cause the CFG to become prohibitively large.

In the FSA in Figure 6, State 1 represents the state in which the ith substep has not been started. When an event "Substep$_i$Started" occurs, the constraint moves into state 2. This event does not appear in the model for the Choice step, but will appear in the model for the substep. State 2 represents the state of the system in which the ith substep has been started at least once. Both states 1 and 2 have transitions on the assertion "Substep$_i$HasNotStarted". In state 1, this transition is a self loop, so encountering this event does not affect the analysis. In state 2, this transition goes to the violation state. FLAVERS treats all of the paths associated with the violation state as infeasible and does not consider them further in the analysis. In this way, the constraint prevents the analysis from considering paths on which a substep is started twice. The transitions "Substep$_i$HasStarted" behave in a similar fashion and prevent



Parallel Step

Try Step

Choice Step

Figure 5: Models of Parallel, Try, and Choice Steps

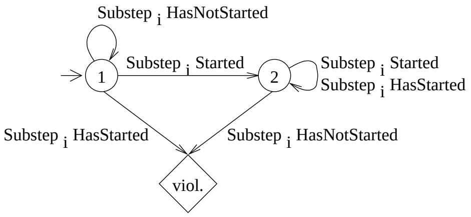


Figure 6: A Feasibility Constraint for the Choice Step

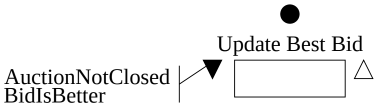


Figure 7: Corrected Step

the Choice step from terminating unless this step has been started. This constraint only deals with substep i, so with n substeps, we may need to use n constraints in the analysis.

The unlabeled node in Figure 5 represents a decision point, where the process can choose between one of its n substeps. From this point, there is a branch representing each choice, guarded by an assertion. If the selected substep completes, the choice step completes. Otherwise, the process moves back to the decision node. Once all substeps have been tried, the process can no longer choose any substep, so the feasibility constraints allow the choice step to terminate by following the branch with n "Substep$_i$HasStarted" guards. The feasibility constraints ensure that the choice step cannot terminate until all substeps have been attempted.

6. EXPERIMENTAL RESULTS

To evaluate our approach to analyzing properties of process definitions, we used FLAVERS to check several properties of the OpenCry Auction. All experiments were run on a Pentium II 400 Mhz PC with 384 MB of memory, running RedHat Linux 5.1 with kernel version 2.0.34. The FLAVERS state propagation algorithm has been written in C and compiled with gcc 2.7.2.3. Currently, we cannot automatically build models directly from Little-JIL process definitions. The purpose of this experiment was to investigate the feasibility of performing analyses on processes. So, for now, we used a combination of manual and automated techniques to generate annotated CFGs, according to the specification of these CFGs as described in the previous section. These CFGs were used to construct the TFG automatically. When constructing a TFG, FLAVERS abstracts away parts of the model that are irrelevant to the property being checked, so the size of the TFG changes depending on the property being evaluated.

As shown in Table 1, the following properties were checked:

- No Late Bids Accepted$_1$: Checks that no late bids can be accepted in the auction.
- No Late Bids Accepted$_2$: Checks the same property, except checked on the process with the "Update Best Bid" step revised as in Figure 7.
- Possible Race Condition: Checks to see if two steps that use "best", which is passed by reference, can be started at the same time. This might be a race condition.
- No Race Condition (no lock): Checks to see if a race condition involing "best" can exist without any locking mechanism in place to restrict access to the paramter.
- No Race Condition (with lock): Checks to see if a race condition involing "best" can exist with a locking mechanism in place to restrict access to the paramter.

7. CONCLUSIONS

This example shows how important it is to apply validation techniques, such as finite state verification, to process definitions. Process definitions are often written at a high level, which allows users to quickly obtain an intuitive understanding of the process. This rapid conveyance of intuition can cause problems by misleading people into incorrect understandings because subtle, yet important,

Property	TFG Nodes	TFG Edges	Result	Time (s)
No Late Bids Accepted$_1$	216	11,837	Inconclusive – fault	6.56
No Late Bids Accepted$_2$	316	30,881	Conclusive	41.10
Possible Race Condition	327	35,788	Inconclusive – fault	143.25
No Race Condition (no lock)	189	7,710	Inconclusive – fault	15.07
No Race Condition (with lock)	269	20,910	Conclusive	17.52

Table 1: FLAVERS Analysis Results

details have been overlooked. The incorrect process shown in Figure 2 was examined by several people who were knowledgeable about both auctions and Little-JIL. Yet it took several days before anyone realized that there was a defect in the process.

We were pleased that the FLAVERS finite state verification system was able to detect this defect and to verify other properties. But this verification was not without problems. Little-JIL uses recursion instead of an explicit looping construct. Finite state verifiers, such as FLAVERS, however, require that recursive constructs be converted to finite representations. The exception handling mechanism of Little-JIL poses still other problems. For example, in a parallel step, more than one substep may generate an exception. If this happens, then the exception handlers can execute concurrently, and the behavior of the process after the handlers finish is dependent on the types of handlers that were executed. Some of the popular features of Little-JIL, such as the choice step, required sizeable flow graphs for their representation, which could lead to increased execution times for FLAVERS' verification. In addition, Little-JIL is a factored language, with the resource manager being a separate component. Certain analyses might require that the control flow of the process and the resource model both be represented. This means that we need to determine a way to represent the resource model for FLAVERS. Feasibility automata may provide a mechanism for doing this, but possibly at the expense of additional complexity and an increase in the time needed for analysis.

In light of this we believe that the constructs in Little-JIL (and by implication other advanced process definition languages) need to be reconsidered in the light of the problems that they may pose for static verification.

8. ACKNOWLEDGEMENTS

The authors would like to thank Aaron Cass, Sandy Wise, and Hyungwon Lee for their help in preparing the example process.

This research was partially supported by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory/IFTD under agreement F30602-97-2-0032, and by the National Science Foundation under Grant CCR-9708184. The views, findings, and conclusions presented here 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, or the U.S. Government.

9. REFERENCES

[1] A. Bröckers and V. Gruhn. Computer-aided verification of software process model properties. In *Proc. of the 5th Int. Conf. on Advanced Information Systems Engineering*, pages 521–546, 1993.

[2] A. G. Cass, H. Lee, B. S. Lerner, and L. J. Osterweil. Formally defining coordination process to support contract negotiations. TR 99-39, University of Massachusetts, Department of Computer Science, 1999.

[3] J. C. Corbett and G. S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6:97–123, Jan. 1995.

[4] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proc. of the ACM SIGSOFT '94 Symp. on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.

[5] C. Fernström. PROCESS WEAVER: Adding process support to UNIX. In *Second Int. Conf. on the Software Process*, pages 12–26, 1993.

[6] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtul-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16(4):403–414, Apr. 1990.

[7] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[8] R. Kadia. Issues encountered in building a flexible software development environment: Lessons from the Arcadia project. In *Fifth ACM SIGSOFT Symp. on Software Development Environments*, pages 169–180, 1992.

[9] M. Kumar and S. I. Feldman. Internet auctions. TR, IBM Institute for Advanced Commerce, Nov 1998.

[10] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic, 1993.

[11] National Institute of Standards and Technology. *Integration Definition For Function Modeling (IDEF0)*, 1993. Federal Information Processing Standards 183.

[12] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proc. of the Int. Conf. Software Engineering*, pages 399–410, 1999.

[13] L. Osterweil. Software processes are software too. In *Proc. of the Int. Conf. on Software Engineering*, pages 2–13, 1987.

[14] A. Wise. Little-JIL 1.0 language report. TR 98-24, University of Massachusetts, Department of Computer Science, 1998.