Verifying Properties of Process Definitions

Jamieson M. Cobleigh, Lori A. Clarke, Leon J. Osterweil
Laboratory for Advanced Software Engineering Research
Department of Computer Science
University of Massachusetts at Amherst
Amherst, MA 01003-6410
+1 413 545 2013
{jcobleig, clarke, ljo}@cs.umass.edu

Abstract

It seems imperative 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. Well-designed process languages provide powerful abstractions for concise and precise specification of such control, but balance this with visualization support to help users also obtain intuitive insights. The underlying complexity of these control abstractions, however, often confounds these 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. Specifically, 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 ecommerce auction example. Although we employ a particular process definition language and analysis technique, our results seem more generally applicable.

Keywords

Process, Formal Methods, Dataflow Analysis, Verification, Auctions, ecommerce

1 Introduction

Processes are pervasive in areas of human and computer interaction, including software engineering processes [11], data mining tasks [10], and work flow models [7]. With the growth of the internet, for example, the role of processes in ecommerce activities, such as auctions, supply chain management, and on-line ordering, has become very prevalent [12, 13, 20]. 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 [19], and that they should be developed, verified, and evolved using approaches that are 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 we continue to be convinced that software processes are software too, we argue that 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, generally visual, representation of common

process notions. 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. We have also developed an interpreter that supports execution of Little-JIL processes. Our experience suggests the need for technology to reason about such process definitions, to assure that their executions conform to important desired characteristics.

In this paper we describe how FLAVERS, 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 and concise 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 and other static analysis techniques.

In the next section we describe a subset of Little-JIL. This subset is chosen so that the reader can follow the example and the ensuing discussion about analysis of Little-JIL process definitions. In section 3 we describe an auction process and present a Little-JIL process definition for it. In section 4 we provide a brief overview of FLAVERS. Like all automated verification systems, FLAVERS must create an accurate model of the computation upon which to base the analysis. FLAVERS builds its model from annotated control flow graph models of the artifacts being analyzed. Section 5 describes the control flow graph model needed to support the Little-JIL constructs. In section 6 we present the results of verifying several properties of the auction example. Section 7 describes how our work builds upon current trends in process languages and software analysis. In the conclusion, we discuss the implication of this work and potential future directions of investigation.

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. It also 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.
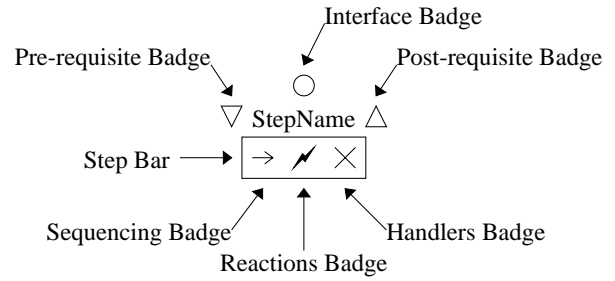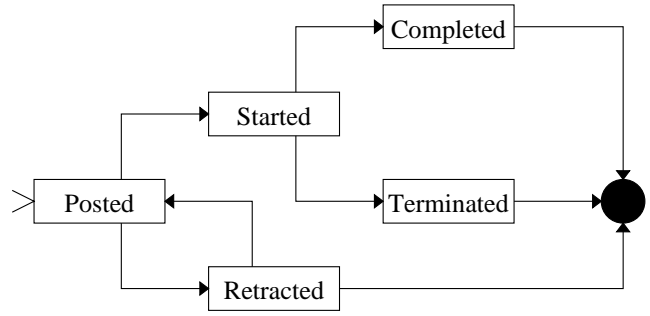

Figure 1: A Little-JIL step

Figure 2: Execution of a Little-JIL step

Steps:
Each step in a Little-JIL definition is represented by a step icon as shown in Figure 1. Each step is given a unique 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. Figure 2 shows the normal flow of control for a Little-JIL step. The step's execution can end when it is in any of the three states that have an arrow pointing to the filled circle. 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. When 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.

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 parent's sequencing badge. All non-leaf steps must have a sequencing badge, which describes the order in which the substeps are performed. Our experience to date indicates that four different substep sequencings are needed for the concise, yet clear definition of processes. They are shown in the key in Figure 3

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.

Requisites:

Process definitions seems to benefit substantially from the attachment of pre- and post-requisites to steps. These constructs are natural vehicles for measuring and monitoring agent performance of steps, and support the retention of process control, while still granting substantial agent latitude and initiative in step performance. 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.

Exception Handling:

There is considerable evidence that processes have extensive and 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, they may have a full hierarchical structure. Our experience also indicates that concise and 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 3.

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 up to its parent. Some handlers may consist only of a badge, but no step structure.

Interface Badges:

Artifact flow and resource specification have both been found to be absolutely essential to the articulate and 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.

A full description of Little-JIL can be found in [21].

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. Of the wide variety of different types of auctions [6, 15], perhaps the most familiar is the Open-Cry Auction, in which a group of people, the bidders, attempt to obtain an item offered for sale by a seller through a real time, interactive process. In the most common type of Open-Cry Auction, the English Auction, 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 to offer. At this time, the high bidder is awarded the item and has to pay the amount 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, and with decreasing amounts of human interaction and scrutiny. For

Sequencing Badges:

$\rightarrow$ Sequential

$=$ Parallel

$\circ$ Choice

$\leftrightarrow$ Try

Handler Control-Flow Badges:

$\uparrow$ Rethrow

$\rightarrow$ Continue

$\sqrt{}$ Complete
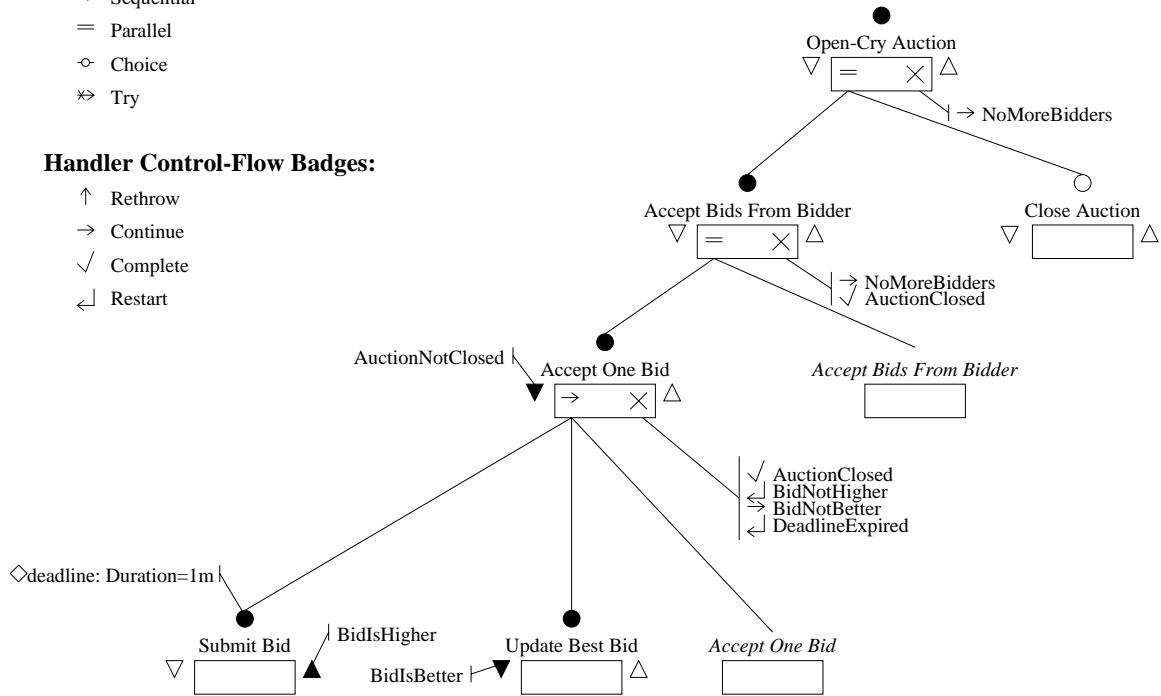
$\hookleftarrow$ Restart

Figure 3: Open-Cry Auction Process

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's role 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 experience of software engineering confirms that analysis of an accurate higher level representation generally offers advantages of scalability and early fault detection. A Little-JIL definition is just such a higher level representation, and it offers the additional advantage of being supported by an interpreter to assure that process execution obeys the same semantics used to verify desirable properties. Little-JIL has many features that make it effective for defining auction processes [3], inviting exploration of the feasibility of verifying properties of such definitions. Figure 3 shows a simplified version of an Open-Cry Auction written in Little-JIL.

At a high level, this process is very straightforward, be-

ing decomposed into two parallel steps, "Close Auction" and "Accept Bids From Bidder". Little-JIL has no direct looping construct, and instead provides support for recursion. Thus, one "Accept Bids From Bidder" step is created for each bidder. Each bidder is handed off to an "Accept One Bid" step, which is responsible for handling all bids made by a single bidder. Because all of these "Accept One Bid" steps can happen in parallel, multiple bidders can be submitting bids simultaneously. All of this bidding happens in parallel with the "Close Auction" step, which the auctioneer performs when it is time to close the auction (presumably after the auction has been open for a fixed amount of time or after no bids have been submitted for a while).

For a bid to be accepted in the auction, the bidder must first submit the bid and then the auctioneer must examine the bid to determine if it is higher than the best bid submitted so far. Then, the bidder can submit another bid for consideration. The "Accept One Bid" step in Figure 3 represents this process for a single bidder. When "Accept One Bid" is started, it first executes a pre-requisite, "AuctionNotClosed" to assure that the bidder cannot submit a bid after the auctioneer has executed the "Close Auction" step. If the pre-requisite completes, then "Submit Bid" is posted for the bidder to execute. If the bidder submits a bid, then a post-requisite, "BidIsHigher" is executed. This requisite ensures that the bid submitted is higher than the bidder's previous bids. If this post-requisite fails, then a "BidNotHigher" ex-

ception is generated. This exception is handled by "Accept One Bid", which has a restart handler causing "Accept One Bid" to restart, giving the bidder a chance to submit a valid bid. "Submit Bid" also has a local parameter named "deadline" of type "Duration" that causes the step to terminate after it has been posted for a specified amount of time, in this case one minute. This generates the exception "DeadlineExpired", which is handled by a restart handler on "Accept One Bid". This deadline is necessary to ensure that the process completes. If this deadline were not here, then if a bidder started a "Submit Step" but never completed it, no ancestor steps would complete and the process would never terminate. When the deadline expires, the "Accept One Bid" step is restarted, and "AuctionNotClosed" is reevaluated assuring that "Accept One Bid", and the entire process, will complete.

If "Submit Bid" completes, then the auctioneer is given "Update Best Bid" to perform. This step's "BidIsBetter" pre-requisite ensures that the update cannot occur unless the bid is better than the best bid submitted, by generating a "BidNotBetter" exception if it fails. This exception is handled by a continue handler at the step "Accept One Bid" which causes the "Accept One Bid" step to be reinstantiated as if the exception never happened.

If the pre-requisite on the recursive call to "Accept One Bid" fails, then it generates an "AuctionClosed" exception that is handled by its parent. The parent has a complete handler for this exception, which causes the parent to complete. When the parent completes, then the entire recursive chain of calls to "Accept One Bid" completes. This allows the "Accept Bids From Bidder" step that spawned this chain to complete when its other substep finishes.

This definition provides a compact, yet clear and precise, representation of a complicated process. The Little-JIL constructs seem effective in supporting intuitive understandings through appropriate process abstractions. Yet, as noted above, these abstractions conceal complex underlying semantics that can mask serious process definition faults. Finite state verification should be used to determine the presence or absence of such faults.

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 [5]. 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 cause 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.

The FLAVERS state propagation algorithm 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 interesting and important properties can be proved by using only a small set of feasibility constraints.

5 Modeling Processes

Earlier we described a representative set of Little-JIL constructs and demonstrated their use in writing a clear, precise, and concise definition of an auction process. In this section we indicate the complexity concealed by some of these constructs by demonstrating their flowgraph models. Moreover, the exception handlers on a step can greatly affect the model for that step. In this section we indicate how some of this complexity can arise. Space does not permit complete specifications of all models of all possible combinations of steps and exception handlers. Rather, we illustrate the models of each step kind using one kind of exception handler, usually one that simplifies the model for that step.
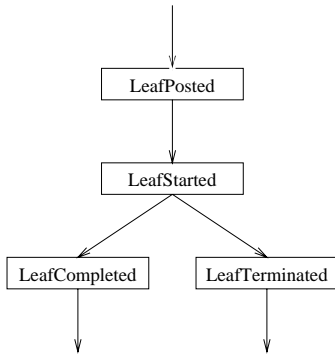
Figure 4: Model of a Leaf Step

Figure 5: Model of a Sequential Step

5.1 Leaf Steps

A Leaf Step in Little-JIL represents a unit of work to be performed by an agent. The model for a Leaf Step is shown in Figure 4. 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 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.

5.2 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 5. 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. Since this is a recursive model, 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.

5.3 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.
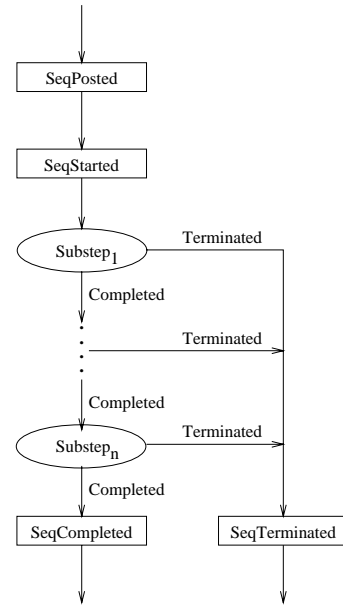
The model of this step, as shown in Figure 6, has a dashed edge in it to represent interactions that may occur due to concurrency. In particular, the dashed edge represents a set of FLAVERS MIP edges, which are used to represent the ways in which flow can move between different tasks. The dashed edge 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[1]. 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 [17].

5.4 Try Steps

Although our Open-Cry Auction process example does not include any choice steps or try steps, we now 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 6, 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.

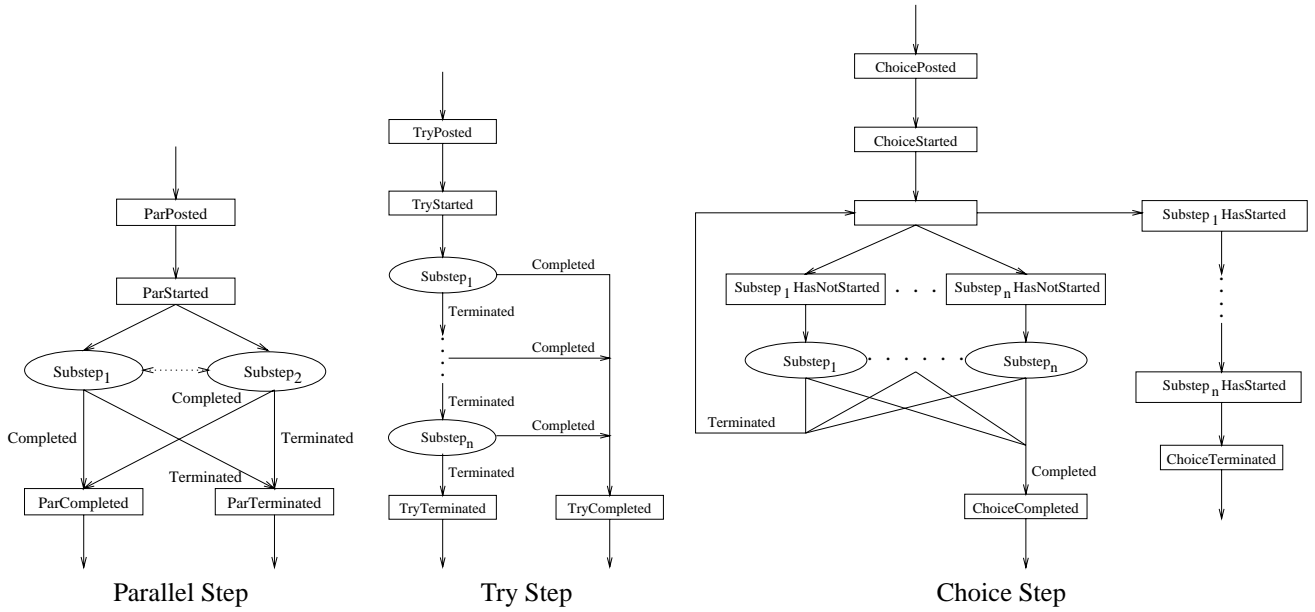[1]In some instances, the substeps may need to be retracted. Modeling this, while possible, exceeds the scope of this paper.

Figure 6 (Models)

Parallel Step

ParPosted → ParStarted → Substep$_1$ ... Substep$_2$

Completed / Terminated → ParCompleted, ParTerminated

Try Step

TryPosted → TryStarted → Substep$_1$ — Completed ; Terminated → Substep$_n$ — Completed ; Terminated → TryTerminated, TryCompleted

Choice Step

ChoicePosted → ChoiceStarted → (decision) → Substep$_1$ HasNotStarted ... Substep$_n$ HasNotStarted → Substep$_1$ ... Substep$_n$; Substep$_1$ HasStarted ... Substep$_n$ HasStarted → ChoiceTerminated ; Terminated ; Completed → ChoiceCompleted

Figure 6: Models of Parallel, Try, and Choice Steps

Figure 7

Substep$_i$ HasNotStarted

1 — Substep$_i$ Started → 2

2: Substep$_i$ Started, Substep$_i$ HasStarted

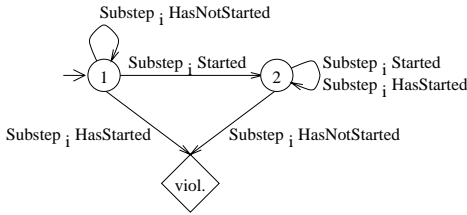Substep$_i$ HasStarted , Substep$_i$ HasNotStarted → viol.

Figure 7: A Feasibility Constraint for the Choice Step

5.5 Choice Steps

Choice steps are similar to try steps, except there is no restriction on the order in which the substeps are attempted; substeps are selected one at a time until one completes. The choice step completes when a substep completes[2]. We assume that all exception handlers in our model of the choice step, shown in Figure 6, are continue handlers. For this model, we have again chosen to use FLAVERS' feasibility constraints to reduce the size of the model. This model has nodes annotated with events, like Substep$_1$ HasNotStarted, that are used in feasibility constraints to limit the ways in which the TFG model is 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. Adding these specially annotated nodes does increase the size of the TFG, but not as much as it would be increased if we were to represent all the ways of selecting the substeps explicitly in the TFG.

The FSA in Figure 7 is a constraint that is used to en-sure that FLAVERS does not consider paths on which the ith substep is started more than once. State 1, the start state, represents the state of the system in which the ith substep has not been started. When an event "Substep$_i$Started" occurs, then 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. When a constraint transitions into its violation state, then FLAVERS treats the path it is currently examining as an infeasible path and does not consider it further in the analysis. In this way, the feasibility 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 the Choice step from terminating unless this step has been started. This constraint only deals with substep i, so for a choice step with n substeps, we may need to use n constraints in the analysis.

The unlabeled node in the model 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 of the substeps have been tried, the process can no longer choose any of the substeps, so the feasibility constraints allow the choice step to terminate by following the branch with n "Substep$_i$HasStarted" guards. The feasibility constraints are

---

[2] As with the try step, retraction of substeps, which exceeds the scope of this paper, may be required.

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

used to ensure that the choice step cannot terminate until all of its substeps have been attempted.

The choice step can add considerable complexity to the model of a Little-JIL process definition. 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 n ways that one of them can be chosen, $n(n-1)$ ways that two of them can be chosen, and so on. This means that in the worst case there are $\mathcal{O}\left((n+1)!\right)$ possible paths that need to be considered. Experience has shown that human agents require the empowerment that constructs such as the choice step provide. The model needed to represent the execution semantics shows, however, that the behavior of the choice step is far from straightforward. This suggests that users may well be attracted to its use, but that reasoning about processes using the choice step is likely to be costly. Our expectation is that human users will greatly benefit from automated aids in reasoning about such processes.

6   Experimental Results

To evaluate our approach to analyzing properties of process definitions, we used FLAVERS to check several properties of the Open-Cry 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, the input FLAVERS expects. These CFGs were used to construct the TFG. 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.

6.1   No Late Bids Accepted

To verify that no late bids can be accepted, it is necessary to verify that on no executions of the process can "Update Best Bid" be started and completed for a bid after "Close Auction" is completed. For this sequence of events, we want
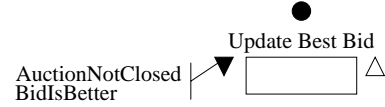

Figure 8: Corrected Step

the "Update Best Bid" that is started and completed to be in the same step. Thus, we need to check this property for each bidder separately. Since each bidder is an instantiation of the same step, we only need to demonstrate that this property holds for one representative bidder.

As shown in Table 1 under the property "No Late Bids Accepted$_1$", FLAVERS returned inconclusive results for this property. Using FLAVERS, we were able to compute an execution trace of a violating path. Upon examination of this path, it was easy to identify the problem with the process. The "Submit Bid" step can be posted before the auction is closed, and started after the auction is closed. There is nothing preventing a bid submitted in this fashion from reaching and being considered by the "Update Best Bid" step. We can correct this by adding an "AuctionNotClosed" pre-requisite in the "Update Best Bid" step, as shown as part of Figure 8. This property with the corrected process was verified conclusively. The details for this are in the row "No Late Bids Accepted$_2$". The reason that the checking this property took longer than the checking of the original version is that FLAVERS is able to stop state propagation as soon as it determines a property is violated. However, with conclusive results, it has to perform state propagation until all possible executions are considered.

6.2   Possible Race Condition

It is possible to use FLAVERS to analyze process definitions to see if there is the potential for race conditions. In order to illustrate this, our Open-Cry Auction model needs to be annotated with some parameter information. The process shown in Figure 9 is still not a complete depiction of the process, but has enough parameter information shown to enable us to check the property.

In Little-JIL, parameter passing is done with a copy-in/copy-out mechanism. For the most part, this prevents multiple steps from using the same variable, which helps reduce the likelihood of race conditions. In our process, how-
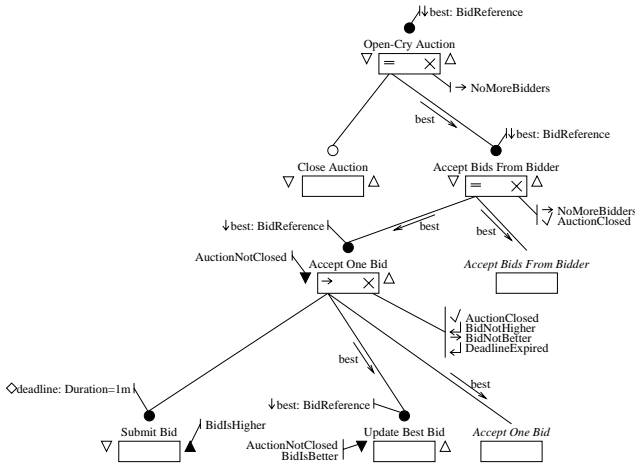
8

Figure 9: Revised Open-Cry Auction Process

ever, there is a parameter "best" of type BidReference. This makes "best" a pointer to a bid, meaning even with copy-in/copy-out semantics several steps can be accessing the bid pointed to by "best" at the same time. This is necessary so the auctioneer, during "Update Best Bid", is able to examine the current high bid at all times. This, however, could also lead to a race condition.

To determine if there is the potential for a race condition in the process, we first need to determine which steps can use the variable "best". This is necessary since "best" is passed into many steps, but only used by a small subset of them. This identification cannot be done directly from the process definition, since it requires knowledge of the agents' behaviors. In this Open-Cry auction we assume that the only step that uses the variable "best" is "Update Best Bid". We can then determine if it is possible for a race condition to exist by checking a property to see if two "Update Best Bid" steps can be started at the same time.

In the Open-Cry Auction, FLAVERS returns inconclusive for this property. By examining the trace provided by FLAVERS, it is clear that two "Update Best Bid" steps for two different bidders can be started at the same time. Since we know now that a race condition can exist, we can take steps to ensure that a failure does not occur from any error in using the variable "best".

6.3 No Race Condition

Suppose there were no protections on the variable "best" to prevent it from being used multiple times simultaneously. Assume that an auctioneer agent is in charge of the "Update Best Bid" step. Then if the auctioneer agent were considering two bids at the same time, it could compare both bids to "best" in any order, but never to each other. As a result, the bid pointed to by "best" at the end of these steps would depend on the order of comparison, and the value of "best"

may not end up being the highest bid in the auction.

Thus, we want to ensure that no access to variable "best" occurs between the time a bid is determined to be better than "best" and the time that "best" is updated to be this new highest bid. FLAVERS was unable to determine conclusively that this property holds if no protections on access to "best" are specified. But, when we incorporated a model the behavior of a lock on variable "best" using a feasibility constraint, we were able to verify the property conclusively. The computation times for both of these checks are shown in Table 1.

We have proven conclusively that if the auctioneer agent locks the variable "best", then this undesirable race condition can be avoided. We have not, however, verified that this process definition does not have a race condition on variable "best". Doing this analysis would require performing analysis on the implementation of the auctioneer agent. Especially for an ecommerce application, it cannot be expected that the analyst will always have access to the implementations of all the agents that will be coordinated by a process definition. If the implementation of the auctioneer were available, then it would be relatively simple to use verification to prove that the auctioneer performs the proper locking and unlocking of the variable "best". Alternatively, we can use feasibility constraints, as we did in this example, to represent the relevant behavior of the missing components. First we would need to show that the process, along with these feasibility constraints is consistent with the specified properties. When the missing agents become available for analysis, we would then need to show that they indeed satisfied the feasibility constraints that we used to model their behavior. This allows us to decompose the analysis process, providing the potential for analyzing large systems that may be distributed among various sites and companies, and written in different languages.

It is important to acknowledge, however, that key components of real world processes may not be available for inspection and analysis. In such cases, verifications of properties such as the one we have been discussing can still be completed and lead to definitive results, but only if credible assurances about their behavior can be provided. It is not unreasonable to envision a future in which participants in processes such as auctions may not wish to provide their componentry, but may be willing to have key properties (such as the one we have been considering) established and certified by disinterested third parties.

7 Related Work

This work expands on the types of analyses that have been explored for process definition systems. For example, the IDEF0 system uses simple forms of analysis to demonstrate certain sorts of well-formedness properties in its diagrammatic definitions [16]. IDEF0 is essentially a dataflow rep-

resentation of a process. As such the analyses performed check for consistent specification of operand inputs and outputs and appropriate nesting. The ProcessWeaver system incorporates stronger semantic features, and uses them for more powerful and diverse sorts of consistency checking [7]. In particular, its greater attention to the specification of operand flows enables ProcessWeaver to perform limited sorts of type checking. In Statemate process definers can develop up to three separate process representations [8]. Statemate performs well-formedness checks on each of the three, but in addition it carries out limited forms of cross-diagram consistency checks.

Perhaps the most ambitious static analysis is carried out in the FunsoftNets system [2]. 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. In addition, such process specific defects as unprocessable object types (which are analogous to dead variable definitions) are also detected.

We are not aware of any other process definition system that has been used as the basis for finite state verification of properties, as described here. However, there is a rich literature describing the application of finite state verification to a wide range of software. Perhaps the work that comes closest to this, is the work on applying verification techniques to architecture description languages (ADL) [1, 18]. ADLs tend to focus on a high level description of system to system communication and on the mechanism for that communication, for example, remote method invocation or event based notification. From one perspective, process languages are similar in that they also provide a high-level language for describing systems of systems. Unlike ADLs, process languages also focus on the interactions of humans with systems. This mix of humans and systems necessitates more complex control flow constructs, including exception handling and reactive control. As a result, dealing with the models of process definitions can be more difficult than the models of ADLs.

Besides FLAVERS, there are several other finite state verification techniques that could be used to verify properties of processes, such as INCA [4], which uses integer necessary conditions, SMV [14], which uses symbolic model checking, and SPIN [9], which uses reachability analysis. We do not believe that any of these would be better able to handle the complex control flow in process definitions but this hypothesis deserves further investigation.

8 Conclusions

The Open-Cry Auction 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, while an advantage, can cause problems by misleading people into incorrect understandings because subtle, yet important, details have been overlooked. The incorrect process shown in Figure 3 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. In doing this, care must be taken to ensure the consideration of event sequences that only happen after deep recursion occurs. 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. Other features of the language may present challenges. For example, 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 this work, representation problems were dealt with by a human, who used ad hoc techniques to translate the Little-JIL definition of the auction process into a suitable input for FLAVERS. Certainly it is desirable for this translation to be done automatically. But the language issues just enumerated will certainly complicate this automation. For all of these reasons 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.

Our success in applying FLAVERS to reason about process definitions suggests that other verification tools and approaches should also be tried. We plan to investigate the use of other static analysis techniques, for example INCA [4], SMV [14], and SPIN [9], to see how they can handle the complexities of analyzing process definitions written in languages such as Little-JIL. Continuation of this research line should lead to better understandings of which verification techniques might be most effective in analyzing which process languages.

9 Acknowledgements

References

[1] R. Allen, D. Garlan, and J. Ivers. Formal modeling and analysis of the HLA component integration standard. In *Proc. of the Sixth Int. Symp. on the Foundations of Software Engineering*, pages 70–79, 1998.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] R. Engelbrecht-Wiggans. Auctions and bidding models: A survey. *Management Science*, 26(2):119–142, Feb. 1980.

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

[8] 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.

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

[10] D. Jensen, Y. Dong, B. S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton Jr., and A. Wise. Coordinating agent activities in knowledge discovery processes. In *Proc. of Work Activities Coordination and Collaboration Conf.*, pages 137–146, 1999.

[11] 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.

[12] M. Kumar and S. I. Feldman. Business negotiations on the internet. TR, IBM Institute for Advanced Commerce, March 1998.

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

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

[15] P. Milgrom. Auctions and bidding: A primer. *Journal of Economic Perspectives*, 3(3):3–22, Summer 1989.

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

[17] 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.

[18] G. Naumovich, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. Applying static analysis to software architectures. In *Proc. of Fifth ACM SIGSOFT Symp. on the Foundations of Software Engineering/Sixth European Software Engineering Conf.*, pages 77–93, 1997.

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

[20] T. Sandholm and V. Lesser. Issues in automated negotiation and electronic commerce: Extending the contract net framework. In *First Int. Conf. on Multi-Agent Systems*, 1995.

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