Verifying Process Models Built Using Parameterized State Machines

Barbara Staudt Lerner
Williams College
lerner@cs.williams.edu

ABSTRACT

Software process and workflow languages are increasingly used to define loosely-coupled systems of systems. These languages focus on coordination issues such as data flow and control flow among the subsystems and exception handling activities. The resulting systems are often highly concurrent with activities distributed over many computers. Adequately testing these systems is not feasible due to their size, concurrency, and distributed implementation. Furthermore, the concurrent nature of their activities makes it likely that errors related to the order in which activities are interleaved will go undetected during testing. As a result, verification using static analysis seems necessary to increase confidence in the correctness of these systems.

In this paper, we describe our experiences applying LTSA to the analysis of software processes written in Little-JIL. A key aspect to the approach taken in this analysis is that the model that is analyzed consists of a reusable portion that defines language semantics and a process-specific portion that uses parameterization and composition of pieces of the reusable portion to capture the semantics of a Little-JIL process. While the reusable portion was constructed by hand, the parameterization and composition required to model a process is automated. Furthermore, the reusable portion of the model encodes the state machines used in the implementation of the Little-JIL interpreter. As a result, analysis is based not just on the intended semantics of the Little-JIL constructs but on their actual execution semantics. This paper describes how Little-JIL processes are translated into models and reports on analysis results, which have uncovered seven errors in the Little-JIL interpreter that were previously unknown as well as an error in a software process that had previously been analyzed with a different approach without finding the error.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking*

General Terms

Verification, Experimentation, Languages

Keywords

Software process, workflow, finite state machine, Little-JIL, LTSA, SMC

1. INTRODUCTION

Software is increasingly being constructed by connecting together existing components into a loosely connected framework. In the past, component reuse was accomplished by the use of component libraries where the result of composition was typically a tightly connected system with the components compiled together into a single executable program. While this is still the most common form of reuse, there is increasing interest in component reuse at a larger scale, where the composite system consists of numerous independent programs running on different machines communicating over a network. Two examples of these types of systems are multi-agent systems and business processes built on top of Web services and other existing applications.

Business processes are typically built by using a process or workflow language that is capable of describing and carrying out coordination between a collection of applications. The process typically defines data flow, control flow, assignment of responsibilities to applications or to humans, and supports decision making and exception handling activities. As business processes become increasingly important to a business's success and increasingly complex, it will become important to determine if these processes behave as expected. In many cases, this involves answering the same sort of questions that are often asked about distributed systems. Is deadlock possible? Are there race conditions? Is starvation possible? As a result, it should be possible to apply the same sorts of analysis tools to business processes as to other distributed applications.

Model checking has been applied to the analysis of concurrent software written in high-level programming languages [4, 20, 6, 10, 19, 16]. One of the major obstacles in verifying programs is in the derivation of an appropriate model that is small enough to analyze yet captures the semantics of the program currently of interest. Process and workflow languages have an advantage over traditional programming languages from this perspective. These languages do not attempt to capture the complete semantics of an application, but rather only define the coordination between external applications or agents. A process written in one of these languages therefore can be more directly translated into a model to be checked in its entirety, but, of course, with the caveat that the external applications are not verified at all using this technique. Nevertheless, model checking of process and workflow languages has the potential for verifying the properties stated directly in the process without the need for a great deal of human intervention in the extraction of a model.

Indeed, various researchers (such as [18, 9, 15, 11, 12, 8]) have applied model checking techniques to the analysis of processes written in a variety of languages, including UML activity diagrams [1], WSFL [13], and BPEL4WS [5]. The technique normally applied is to translate the process language into the input language of a model checker and then apply the model checker to do the analysis. These process languages contain a similar set of control constructs: sequential, choice, parallel, and iteration. In this previous work, exception handling was omitted from the features of the language that were modeled. Exception handling complicates model checking due to its unpredictability and the non-local control flow that ensues. Nevertheless, it is an important feature of process languages, and, particularly due to the difficulty of informally reasoning about the impacts of exception handling, it is critical to include it in the models for verification.

In this paper, we describe how we use LTSA to model check processes written in the Little-JIL process language. Processes written in Little-JIL are hierarchical decompositions of steps into substeps and exception handling steps. Model checking is accomplished by translating Little-JIL steps (including exception handling) into FSP and then composing the FSP specifications to produce a model of an entire process. We take advantage of the hierarchical nature of the processes by minimizing the generated labeled transition system at each level of the hierarchy. This has proved effective in keeping the models to a tractable size.

Previous work on model checking process languages focused on language semantics. While sensible, this assumes that the language semantics will be implemented correctly in the execution support for the language. This is not an issue for UML's activity diagram notation, which is purely a modeling language, but is an issue for BPEL4WS, which is an executable language. Some of these control constructs require careful implementation to be faithful to the language semantics because the execution of these processes are expected to be distributed across machines and to be carried out by external entities. In this paper, we show how the models we produce incorporate a model of the Little-JIL interpreter. As a result, we are able to prove properties not just about the process specifications, but also about their execution using the interpreter. In this paper, we report on how we derive the models used in model checking from the process and interpreter implementation. We also report on the analysis results we have achieved thus far, including the uncovering of errors in the interpreter implementation and one of the processes we have analyzed.

In the remainder of this paper, we first give an overview of Little-JIL and then an overview of LTSA. We next describe how we build models of Little-JIL processes and then describe some results we have obtained through analysis. We conclude with future work.

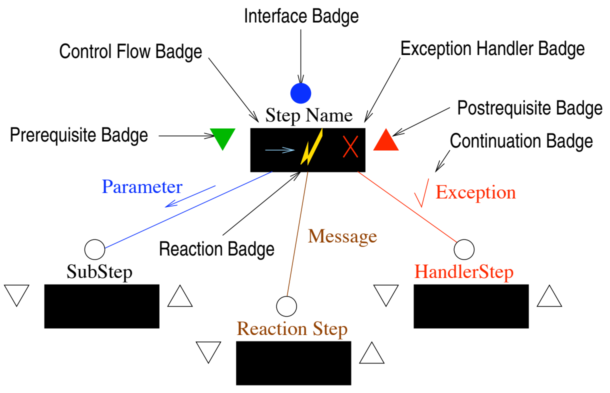

Figure 1: Graphical Syntax of a Step

2. LITTLE-JIL

Little-JIL is a process language with a visual syntax [22]. Little-JIL processes describe the coordination of activities carried out by external entities. These external entities may be people, Web services, intelligent software agents, or any other external component that is capable of carrying out requested tasks on input data and reporting the results, whether successfully completed or incomplete due to exceptional situations. We refer to these external entities as *agents*, independent of their actual realization.

A Little-JIL process is executed by interpreting the process specification, selecting agents to perform the tasks, informing the agents of their tasks, sending the necessary data to those agents to carry out their tasks and accepting the return results. Different agents may participate from different machines and may change which machine they are operating from over time. Agents may be requested to perform tasks concurrently. Thus, processes written in Little-JIL are typically executed as distributed, concurrent systems.

In this section, we present the aspects of Little-JIL relevant to the model checking analyses we have performed and provide more details on those aspects of the interpreter implementation that are relevant to the model checking.

2.1 The Little-JIL Process Language

The analyses we describe in this article pertain to control flow. As a result, we will describe the control flow semantics of Little-JIL and omit other details of the language. (For a complete description of the language, please see the Little-JIL Language Report [21].)

A process is defined as a collection of hierarchically-decomposed *steps* as shown in Figure 1. A step defines an activity to be performed. It is connected to other steps via substep edges and exception handling edges. Non-leaf steps represent high-level activities that are decomposed into finer-grained activities using substep edges. Leaf steps represent activities that are carried out by external agents with no further work decomposition specified, leaving the agent free to choose exactly how to perform the activity.

Non-leaf steps use *sequencing badges* to define the control flow among the substeps. There are four sequencing badges as shown in Figure 2. *Sequential* steps perform their substeps sequentially from left to right. *Parallel* steps assign their substeps to agents concurrently. The agents might

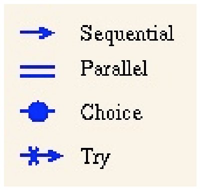

Figure 2: Sequencing badges of Non-Leaf Steps

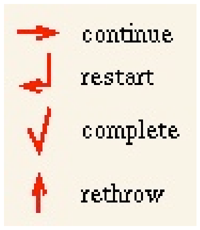

Figure 3: Exception Handling Control Flow

actually perform the work concurrently or in some unspecified sequential order. *Choice* steps assign substeps to agents concurrently, but allow only a single agent to perform the assigned work. As soon as one agent begins a choice substep, all other choice substep assignments are retracted. *Try* steps assign work to agents sequentially but stop as soon as one of the substeps completes successfully. A substep may be marked as optional, meaning that the agent may decide to skip the step at runtime.

If a step cannot be completed successfully, it terminates and throws an exception. The parent of the step can attempt to handle the exception. *Exception handling* activities are attached to non-leaf steps via exception handling edges. The edges specify which exception the step handles and where control should flow if the exception is successfully handled. There are four possibilities for control flow following an exception as shown in Figure 3. A *continue* handler indicates that the step should be continued from the point at which the exception was handled. A *restart* handler indicates that the step containing the exception handler should be restarted from the beginning. A *complete* handler indicates that the step containing the exception handler should be considered to have completed successfully. A *rethrow* handler indicates that the exception should be rethrown to the parent of the step containing the exception handler. It is possible for an exception handler to have a null step; that is, it identifies the exception being handled and the control flow semantics to use but does not involve any activity to compensate for the exception.

Steps may also have *prerequisites* and *postrequisites.* Requisites are themselves steps. A prerequisite uses the state of the process to ensure that it is appropriate to execute the step. It is executed prior to a step's substeps. The substeps are executed only if the prerequisite completes successfully. A postrequisite is intended to check the results of the substeps and only allow the step to complete and make its results visible if this check is passed. A postrequisite is executed after a step's substeps are completed. Both prerequisites and postrequisites may be omitted from a step definition.

Every step is assigned to an *agent.* The agent is the external entity responsible for the step. It is the agent that decides when to start a step and when to skip an optional step. Deciding when to start a step is particularly important for choice substeps as it results in the retraction of the other choice substeps. For leaf steps, the agent is responsible for performing the work associated with the step, returning data results on successful completion of the work, and throwing an exception on unsuccessful completion.

Figure 4 shows the process used in the experiment described in this paper. This process coordinates the actions of an auctioneer and a collection of bidders interacting in an open-cry auction. The auctioneer is responsible for accepting bids and deciding when to close the auction. The bidders are responsible for submitting bids. The `Close Auction` step runs in parallel with the `Accept Bids From Bidder` step, meaning that the auctioneer can decide to close the auction at any time. `Accept Bids From Bidders` is a recursively parallel step. This allows it to accept bids from different bidders in parallel. The `Accept One Bid` step is a sequentially recursive step. It first checks that the auction is still open via a prerequisite. If it is open, it allows a bidder to submit a bid. If it is the best bid seen so far, the auctioneer updates its record of the best bid. Whether or not the new bid is best, the bidder is then allowed to bid again via the recursive `Accept One Bid` step if the auction is still open.

2.2 The Little-JIL Interpreter

The Little-JIL interpreter uses a collection of interacting finite state machines that track the status of the steps. These finite state machines also form the core of the models used in model checking processes. The most important state machines for a step are a `StepInterpreter` and a `Sequencer`. The `StepInterpreter` captures the semantics of step execution that are independent of the sequencing badge while the `Sequencer` captures the semantics that vary depending on the sequencing badge. There are 5 types of Sequencer machines, one for each sequencing badge plus one for leaves. In total, there are 13 types of state machines within the interpreter. In addition to the major ones listed above, there are machines that interact with the resource manager, evaluate prerequisites and postrequisites, instantiate a step, handle exceptions, and clean up a step when it completes.[1]

The interpreter state machines are defined using SMC[2] (State Machine Compiler) which generates executable Java code from the state machine descriptions. For example, Figure 5 shows the state machine for the sequential sequencer. Figure 6 shows the definition of the `INSTANTIATING_SUBSTEP` state of the `SequentialSequencer` state machine. On entry to this state, if there are more substeps, the next substep is instantiated. This method call creates a new thread running the `StepInterpreter` machine (not shown) for the substep. If there are no more substeps, the `SequentialSequencer` machine completes, ending execution of the nested state machine, and sending the `sequencerCompleted` event to the `StepInterpreter` machine it is nested inside of. After calling `instantiateNextSubstep`, the sequencer expects to hear either the event `substepInstantiationSucceeded` or `substepInstantiationFailed`. In the succeeded case, it exe-

[1]There is an additional machine to handle reactions, but that is not included in the experiment reported here.
[2]http://smc.sourceforge.net/

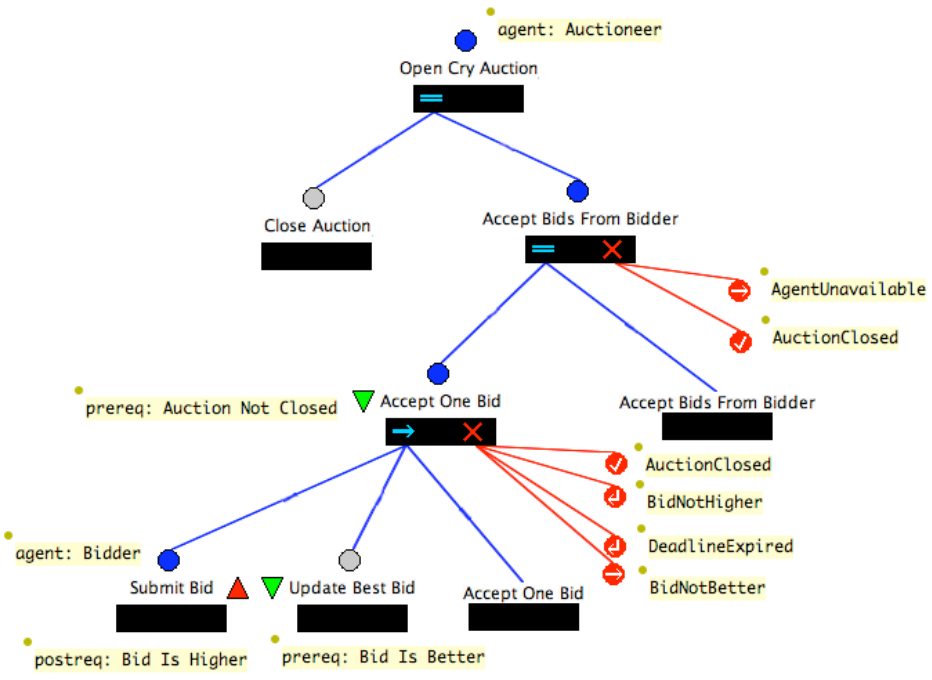

Figure 4: Auction Process

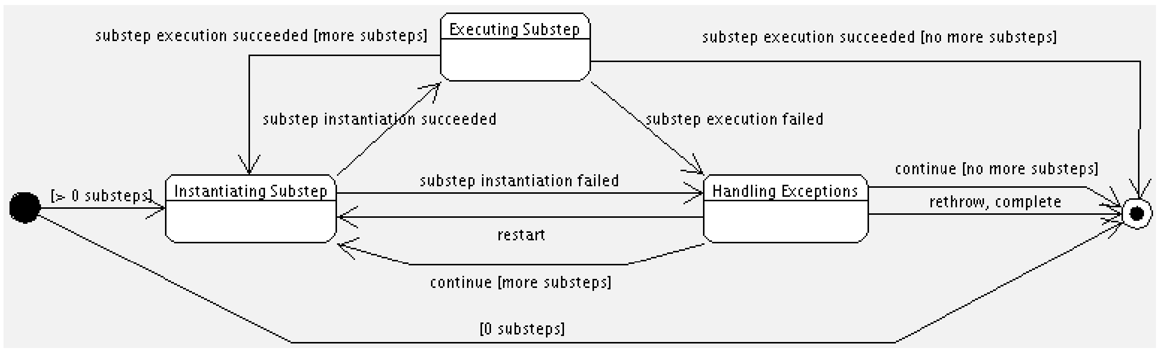
substep execution succeeded [more substeps]
Executing Substep
substep execution succeeded [no more substeps]
substep instantiation succeeded
substep execution failed
continue [no more substeps]
[> 0 substeps]
Instantiating Substep
substep instantiation failed
Handling Exceptions
rethrow, complete
restart
continue [more substeps]
[0 substeps]

Figure 5: The State Machine of a Sequential Sequencer

```
INSTANTIATING _SUBSTEP
Entry{start();}
{
    start[moreSubsteps();]
        nil {instantiateNextSubstep();}
    start[!moreSubsteps();]
        pop(sequencerCompleted) {}

    substepInstantiationSucceeded(substep: InterpreterAgendaItem)
        EXECUTING_SUBSTEP {postSubstep(substep);}

    substepInstantiationFailed(substep: InterpreterAgendaItem, exceptions: Collection)
        HANDLING_EXCEPTIONS {addExceptionsToHandle(exceptions);}
}
```

Figure 6: The INSTANTIATING_SUBSTEP State of the Sequential Sequencer in SMC Syntax

```
CLOCK = (tick -> tock -> CLOCK).

CHIME(MaxValue=10) = COUNT_SECONDS[0],

COUNT_SECONDS[seconds:0..MaxValue-1] =
 (when seconds < MaxValue-2
     tock -> COUNT_SECONDS[seconds+2]
| when seconds >= MaxValue-2
     tock -> chime -> COUNT_SECONDS[0]).
```

Figure 7: Example FSP Process

```
||MINUTE_ALARM = (CLOCK || CHIME(60))
     @{chime}.

CHECK_MAIL =
    (chime -> check_mail -> CHECK_MAIL).

||CHECK_MAIL_EVERY_MINUTE =
    (MINUTE_ALARM || CHECK_MAIL).
```

Figure 8: A composite process in FSP

cutes the action to post the substep, causing the substep's `StepInterpreter` to receive the `posting` event, and then transitions to the `EXECUTING_SUBSTEP` state. If the substep instantiation fails, the sequencer remembers what exception needs to be handled and then transitions to the `HANDLING_EXCEPTIONS` state.

The exact behavior of a state machine may depend on static information about a step as well as dynamic decisions made by an agent. For example, the condition `moreSubsteps()` used in Figure 6 depends on the static information of how many substeps the current step has as well as which was the last substep to complete. Information that is only known dynamically includes information about whether an agent completed the work of a leaf step successfully or not.

3. LTSA

LTSA is a model checker developed by Jeff Kramer and Jeff Magee at Imperial College [14]. The input to LTSA is a set of interacting finite state machines written in the process algebra FSP (Finite State Processes).[3] Each FSP state machine definition is translated into a Labeled Transition System for analysis by LTSA. A state machine is defined in terms of states consisting of a list of transition choices. Each transition may have a condition associated with it, followed by a list of sequential events, and ending with the name of another state. When two or more state machines use the same event, the event is assumed to happen synchronously in all state machines.

Figure 7 shows two interacting FSP state machines. The first state machine simulates a clock that issues alternating ticks and tocks. A second state machine called `CHIME` issues a chime periodically where the length of the period is determined by the value passed in as a parameter to `CHIME`. Using parameterization, the same process can be used in different contexts and result in the generation of different labeled transition systems. In this case, the `CHIME` labeled transition system requires enough states to count by 2 up to its maximum value.

State machines as shown in Figure 7 are primitive. Primitive state machines may be composed to produce composite state machines. When composing state machines, it is possible to hide some of the events used by the primitive state machines and only make visible those required for later synchronization with other state machines. This allows the composite state machine to be minimized, thereby reducing the size of the labeled transition systems that must be analyzed. This helps address problems of state space explosion common in model checking and is a good approach for modeling systems with a hierarchical organization [2, 23].

Figure 8 shows an example composite state machine. The `MINUTE_ALARM` state machine is defined by composing `CLOCK` and `CHIME`, passing 60 as the value to count to, and exposing only the chime event. `CHECK_MAIL` models a mail client that checks the mail server for incoming mail whenever the chime goes off. The tick and tock events are not exposed by the `MINUTE_ALARM` composite state machine. This allows the composite state machine `CHECK_MAIL_EVERY_MINUTE` to be minimized to a finite state machine consisting of just two states. If `MINUTE_ALARM` exposed the tick and tock events, the composite state machine `CHECK_MAIL_EVERY_MINUTE` would contain 124 states. By hiding the tick and tock events, the composite state machine can be minimized to a 2-state machine equivalent to the `CHECK_MAIL` state machine itself.

After composing a model, LTSA can determine whether there is deadlock in the model by looking for states in the composite state machine with no outgoing edges. In addition, an analyst can define properties for LTSA to check. A property is defined as another finite state machine specified in FSP, but specially marked as being a property. After composing a property with other finite state machines, LTSA can determine whether the property holds for the composite state machine. If LTSA finds deadlock or determines that a property does not hold, it produces a trace of events through the finite state machine that demonstrate the failure.

4. BUILDING MODELS OF LITTLE-JIL PROCESSES FOR MODEL CHECKING

We have two goals in building models of Little-JIL processes. The first goal is precision: the model must represent the process with enough precision that analysis can produce interesting results. The second goal is automation: the amount of manual effort required to build a model for a Little-JIL process and analyze it must be minimal. We have accomplished these goals by dividing the Little-JIL model of a process into two parts: a reusable part that precisely captures the language semantics and a process-specific part that is automatically generated. The reusable portion was created manually by translating the state machines implemented in the interpreter into FSP. We parameterized the FSP descriptions in the reusable portion so that they can be specialized to the properties of individual steps in a Little-JIL process. Then, we built a tool that generates the process-specific part by composing and parameterizing state

[3]Unfortunately, both Little-JIL and FSP use the word process but mean very different things. To avoid confusion, we use state machine terminology when referring to the FSP descriptions rather than the process algebra terminology suggested by Kramer and Magee.

```
INSTANTIATING_SUBSTEP[curSubstep:1..NumSubsteps+1] =
      (when curSubstep < NumSubsteps+1
             substep[curSubstep].requestInstantiate -> INSTANTIATING_SUBSTEP[curSubstep]

      | when curSubstep < NumSubsteps+1
            substep[curSubstep].instantiationSucceeded -> substep[curSubstep].requestPost ->
                EXECUTING_SUBSTEP[curSubstep]

      | when curSubstep < NumSubsteps+1
            substep[curSubstep].instantiationFailed -> HANDLING_EXCEPTIONS[curSubstep]

      |  when curSubstep == NumSubsteps+1 parent.sequencerCompleted -> FINAL
)
```

Figure 9: The INSTANTIATING_SUBSTEP Local Process of the Sequential Sequencer in FSP

machines from the reusable portion to construct a model for each step. The models of non-leaf steps additionally compose models of their substeps, using event renaming to create the necessary synchronizations between steps and their substeps. In this section, we explain how this model is constructed.

4.1 Modeling the Interpreter

The major effort in modeling a Little-JIL process is in modeling the interpreter. Due to the state machine based implementation of the interpreter, it is relatively straightforward to translate the interpreter's state machines into an FSP. Each FSP state machine is parameterized so that it can perform conditional tests equivalent to those performed by the interpreter's state machines. Actions that result in the sending of events between state machines, such as posting a substep, are modeled by adding actions to the FSP descriptions to represent the sending and receiving of those events.

For example, consider the FSP equivalent of the INSTANTIATING_SUBSTEP state shown in Figure 9. The INSTANTIATING_SUBSTEP FSP state is parameterized by the index of the substep currently being executed. The SEQUENTIAL_SEQUENCER state machine (not shown) of which this is a state has NumSubsteps as a parameter. The model for each sequential step includes a SEQUENTIAL_SEQUENCER state machine in its composition, passing in the number of substeps it has as a parameter. Note that by using the indexing feature of FSP, it is possible to describe this state machine independently of the number of substeps a step has. When a specific step is modeled, a parameter will indicate how many substeps the step has, allowing the appropriate labeled transition system to be generated from this description. The requestInstantiate action models the activities of the instantiateNextSubstep method. The requestPost action models the activities of the postSubstep method.

Each time that a state machine starts in the interpreter, it is run in a new thread. To model this, each step instantiates its own FSP state machine by supplying the appropriate parameter values. In addition to these state machine threads, four actions taken by the state machines result in the creation of new threads. Two appear in the example above: step instantiation and step posting. The FSP model includes models of these threads as well. Figure 10 shows the model for the thread that instantiates a step. The requestInstantiate event synchronizes with the FSP

```
STEP_INSTANTIATOR =
      (requestInstantiate -> instantiate ->
            STEP_INSTANTIATOR

     | final -> STEP_INSTANTIATOR)
```

Figure 10: Model of the Thread that Instantiates a Step

state machine doing the instantiation, in this case the sequencer for the parent of the step being instantiated. The instantiate event synchronizes with the StepInterpreter process for the step being instantiated. FSP state machines for the other three machines are quite similar but involve different events.

The FSP model of the interpreter contains 18 processes corresponding to the 13 interpreter state machines[4] plus the 4 helper threads mentioned above. In addition, the FSP model contains a process describing agent behavior, specifically placing the agent in charge of deciding when to start a step, when to opt out, when to complete successfully, and when to throw an exception. In total, there are 107 states in the FSP description, compared to 68 in the interpreter implementation. There are more states in the FSP definition than states in the SMC definition for several reasons. The FSP definition contains 6 additional state machines (the agent, the sequencer for parallel steps with 0 substeps, and 4 helper threads). The FSP definitions have an additional FINAL state. In some cases it was necessary to add states to model activities that were done in Java actions. These primarily revolved around throwing and handling exceptions. Finally, in the case of the choice sequencer, it was necessary to replicate an entire state for each child of the substep, not just replicate transitions as shown earlier.

In summary, the FSP model was built directly from the implementation. While including somewhat more functionality than just the SMC definitions, the connection between the two representations is very clear and represents a faithful model of the interpreter implementation. The very direct relationship between the model and the interpreter implementation increases our confidence that the model is accurate and leads to analysis results that are able to find errors

[4]The parallel sequencer requires a separate state machine for the special case in which a parallel step has 0 substeps.

in the interpreter implementation and provide more detailed analysis of processes.

4.2 Modeling a Little-JIL Process Using FSP

To model a Little-JIL process, we create a model of each step using the step name to distinguish one step model from another. We provide parameter values that capture the static information that the model depends on about the step, such as the number of substeps a step has, or whether the step is optional. We then compose the step models, renaming parent and substep action prefixes to the appropriate step names to create a model of the entire Little-JIL process.

Each step model is itself a composition of primitive FSP state machines. In particular, each step is modeled with a `StepInterpreter`, a `Sequencer`, an `Elaborator`, a `Finalizer`, an `Agent`, a `StepInstantiator`, and a `StepPoster`. Depending on the characteristics of the step, additional primitive state machines, such as a `PrerequisiteEvaluator` or `ExceptionHandler` may also be included. Figure 11 shows the model generated automatically for the `OpenCryAuction` step of Figure 4. The parameters provide information to the primitive state machines about the details of the step. For example, the parameters to the `STEP_INTERPRETER` indicate that the step acquires resources, has no prerequisite, is a parallel step, is the root of the process and has no postrequisite.

Figure 11 also shows how a step model is decomposed into a `STEP_INTERPRETER` machine and a `SEQUENCER` machine. The `SEQUENCER` machine is responsible for coordinating with the child steps. The parameter to the `PARALLEL_SEQUENCER` machine composed into the `OpenCryAuction_Sequencer` machine indicates that the step has two children. Action prefix renamings are used to rename the generic `parent` and indexed `child` action prefixes used in the reusable model to the steps that serve as the actual parent and children for this sequencer. In this case, `openCryAuction` is identified as the parent of `close_Auction` and `accept_Bids_From_Bidder`, the prefixes used to identify the corresponding steps in the FSP model.

Other steps are composed in a similar way. Recursive steps require special attention. Since the model must be finite, it is necessary to remove recursion from the model. Currently, we require the analyst to modify the Little-JIL process to unroll recursion the desired number of times before terminating it. This version with the recursion eliminated can then be translated using our tool.[5]

Using the approach described above, we have used our tool to translate the auction process shown in Figure 4 into an FSP model. We first removed recursion by eliminating all recursive calls by hand, although it would be possible to model recursion by manually unrolling the recursion the desired number of times prior to translating to FSP. Table 1 shows the sizes of the models created by LTSA before and after minimization for each step in the auction process shown in Figure 4. Composition time indicates how long it took to compose the model for the step, while minimization time is the time to minimize the model after it is composed. Times are measured in seconds and were taken on a Dell Dimension 4100 running Red Hat Linux 7.1 with 512 MB of memory, running Java 1.4.2. The times are based on a model of the interpreter that corrects the errors reported in Section 5, resulting in no deadlocks or violated properties. The time to check the model of the OpenCryAuction once the model is created is 3 ms.

[5]We are working on a tool that allows the analyst to specify the number of recursion unrollings to include, along with other customizations prior to analysis. This is discussed further in Section 6.

5. PROVING PROPERTIES OF LITTLE-JIL PROCESSES

After building the FSP model of the Little-JIL auction process, we then used LTSA to check for deadlock and to check properties about the interpreter implementation. Our analysis with LTSA has revealed seven previously unknown errors in the interpreter implementation. Two of these could have been uncovered given the right test input, while the remaining five would only be apparent if a particular interleaving of events from concurrent threads occurs. We find this result to be particularly exciting given that these errors never surfaced in the testing or routine use that the interpreter has undergone in the past several years.

We next modified the generated model to add process-specific properties to duplicate an experiment using FLAVERS [7] to analyze the auction process [3]. We duplicated three properties from the previous experiment. The other two properties from the previous experiment included a detailed model of agent behavior, which cannot be generated from the process description alone and thus is not included in this experiment. We agreed on the results of two properties but disagreed on the third. On further manual examination the results of the current analysis are correct. Thus, we have uncovered an error in the auction process that was not discovered by the earlier analysis with FLAVERS.

5.1 Interpreter Deadlocks Found

Most of the errors that were found were uncovered during deadlock detection. One design problem is responsible for three of these errors so we will discuss those in more detail. All substeps of a parallel step may be executed concurrently. The interpreter posts them concurrently, but agents are responsible for deciding exactly when they execute. If a substep of a parallel step throws an exception, any other substeps that have not been started yet by an agent are retracted (preventing the step from starting) while the exception is being handled to avoid doing work that is potentially unnecessary. Retracting a step is done using a helper thread similar to step instantiation and posting:

```
STEP_RETRACTER =
    (requestRetract -> retracting -> STEP_RETRACTER
   | final -> STEP_RETRACTER).
```

The Sequencer of the parent synchronizes with this thread using a `requestRetract` event. The `StepInterpreter` of the child synchronizes with the `retracting` event. Because retraction is done in a separate thread there can be an arbitrarily long time between when the parent sequencer requests the retraction and the substep actually retracts. This sets up the possibility of races, some of which result in deadlock:

- If the parent's finalizer starts (meaning the step is cleaning up) before the substep retracts, the substep never gets canceled as it should be.

```
minimal
||OpenCryAuction =
    (
        openCryAuction:STEP_INTERPRETER(
            True,    // Acquires resources
            False,    // Does not have prerequisite
            0,    // Number pre/postreq exceptions
            Parallel,
            True,    // Is root
            False)    // Does not have postrequisite
        || {openCryAuction}::STEP_POSTER/{final/openCryAuction.final}
        || {openCryAuction}::RESOURCE_ACQUIRER(1,  // Number of non-agent resources
            False) // Do not force a resource acquisition to fail
        || {openCryAuction}::RESOURCE_RELEASER(2)  // Number of resources
        || {openCryAuction}::ELABORATOR(1,  // Number of non-agent resources
                True)    // Acquires an agent
        || {openCryAuction}::FINALIZER (2,    // Number of resources to release (approximate)
                2)    // Number of substeps
            /{
                close_Auction/openCryAuction.child[1],
                accept_Bids_From_Bidder/openCryAuction.child[2]
            }
        || {openCryAuction}::AGENT (Parallel,
                False,    // Not optional
                False,    // Does not throw exceptions
                False,    // Does not have deadline
                False)    // Does not depend on global
        || OpenCryAuction_Sequencer
    )
}.

minimal
||OpenCryAuction_Sequencer =
    (
        PARALLEL_SEQUENCER (2)    // Number of children
            /{
                openCryAuction/parent,
                close_Auction/child[1],
                accept_Bids_From_Bidder/child[2]
            }
        || {close_Auction}::STEP_RETRACTER/{final/close_Auction.final}
        || {accept_Bids_From_Bidder}::STEP_RETRACTER/{final/accept_Bids_From_Bidder.final}
        || {close_Auction}::STEP_CANCELLER/{final/close_Auction.final}
        || {accept_Bids_From_Bidder}::STEP_CANCELLER/{final/accept_Bids_From_Bidder.final}
        || Close_Auction/{openCryAuction/close_Auction.parent}
        || Accept_Bids_From_Bidder/{openCryAuction/accept_Bids_From_Bidder.parent}
        || OpenCryAuction_ExceptionHandler
   )
}.
```

Figure 11: The FSP Model of the CloseAuction Step

	Before minimization		After minimization		Composition	Minimization
Step	States	Transitions	States	Transitions	time (sec)	time (sec)
OpenCryAuction	195	293	22	28	0.3	0.07
CloseAuction	92	175	13	25	0.3	0.02
AcceptBidsFromBidder	154	287	16	29	0.3	0.04
AcceptOneBid	174	349	27	45	0.3	0.06
SubmitBid	126	221	17	31	0.3	0.05
Update Best Bid	114	215	16	29	0.3	0.03
Total					1.8	0.27

Table 1: Sizes of the Models of the Steps in the OpenCryAuction process and Time to Create Them

- The exception handler of parallel steps assumes that steps retract before the exception handler runs, but there is no synchronization in the interpreter to assure this. This leads to two problems:
  - A substep may start while the parent's exception handler is running, which is in violation of language semantics.
  - If an agent opts out of a substep after the parent's exception handler starts, the exception handler deadlocks.

5.2 Violations of Interpreter Properties

We defined nine properties that we wanted to prove about the interpreter. These properties include such things as assuring that the prerequisite is checked prior to executing substeps. Of these nine properties, two uncovered errors in the interpreter. One property is again related to substep retraction so we discuss it here. This property states that a sequential or parallel step can complete successfully only if one of the following holds:

- All children complete successfully, or
- All exceptions thrown by substeps are handled with complete or continue control flow semantics.

This property was included in the composition for all parallel and sequential steps. In attempting to prove it, we discovered another race condition involving retraction of substeps of parallel steps. Specifically, if the parent's exception handler runs and decides to continue the step, it will think there is no more work to do and complete the step if the substeps that should be retracted are not yet retracted.

5.3 Process Errors Found

To prove process-specific properties, those properties must currently be defined manually. In addition, minor modifications to the generated model must be performed manually to expose the actions used in the property from the state machines where they are defined to the state machine where the property can be proven. In this section, we describe the four changes that were made to prove that it was possible for bids to be submitted after the auction is closed.

The first change required was to relate the `CloseAuction` step with the `AuctionNotClosed` prerequisite of the `Update-BestBid` step. The action performed by the agent of the `CloseAuction` is to set a variable to indicate that the auction is closed. The action performed by the agent of the `Auction-NotClosed` step is to check the value of this variable. The `Agent` state machine included in the reusable portion of the model has a parameter to control whether global state affects the agent's behavior, and, if so, takes the value of that global into consideration when deciding whether to complete successfully or fail. To model the `auctionClosed` variable, the following was done to the model of the `AuctionNotClosed` step:

- The parameter indicating that this step's agent uses a global is changed to true (see Figure 12).
- The `globalSet` action of this step's agent is renamed to `auctionClosed` so that it can synchronize with other state machines that set this variable (also shown in Figure 12).

```
{auctionNotClosed}::AGENT (Leaf,
    False, // Not optional
    True,  // Can throw exceptions
    False, // Does not have deadline
    False, // Do not force failure
    True)  // Does depend on global -
           //  manually changed
     /{
        // Manually added renaming
        auctionClosed/auctionNotClosed.globalSet
     }
```

Figure 12: The Modified Agent to model the AuctionClosed global

- The `auctionClosed` action is added to the list of actions to make visible from the `AuctionNotClosed` composite state machine.

In addition, the following was done to the model of the `CloseAuction` step:

- The `setGlobal` action of this step's agent is renamed to `auctionClosed` so that it can synchronize with other state machines that look for this variable being set.
- The `auctionClosed` action is added to the list of actions to make visible from the `CloseAuction` composite state machine.

Second, the property to be proven must be defined manually. While this requires understanding the events used by the model, many properties can be expressed concisely. The property under consideration here, `NoLateBidsAccepted` is defined as:

```
property NO_LATE_BIDS_ACCEPTED =
    ( update_Best_Bid.completed ->
            NO_LATE_BIDS_ACCEPTED
    | auctionClosed -> CLOSED),

CLOSED =
    ( auctionClosed -> CLOSED).
```

Third, the property must be composed with the least common ancestor of all involved substeps. In this case, this requires composing the property with the model for the `OpenCryAuction_Sequencer`.

Finally, all events that are required by the property must be visible from the originating state machine to the state machine in which the property is being checked. In some cases, they may already be visible. In other cases, they must be added to the list of events that are made visible.

Note that these changes all affect parameterization of state machines, renaming of events, and exposing events previously hidden. All of the changes are made in the compositions used to model the actual process steps. The reusable portion of the model that captures the language semantics is not modified at all. This further supports the appropriateness of the approach of using a reusable language semantics model that is customized to capture the semantics of a particular process.

In the experiment referenced earlier and in the analysis experiment described here, the result was that the `NO_LATE_BIDS_ACCEPTED` property does not hold. The `CloseAuction` step can close the auction after the AuctionNotClosed prerequisite is executed and before the UpdateBestBid step is executed, leading to the property failure.

Based on this result, in the previous experiment, the authors modified the process so that the UpdateBestBid step would check the AuctionNotClosed prerequisite again, in addition to the BidIsBetter prerequisite it currently had. The authors reported that the NO_LATE_BIDS_ACCEPTED property could be proven for this modified process. With the more detailed model of language semantics provided in this experiment, however, we can prove that this property still does not hold. In particular, it fails if the CloseAuction step executes after UpdateBestBids's prerequisite but before UpdateBestBid completes. For the property to be proven, it is necessary for the AuctionNotClosed prerequisite and the UpdateBestBid step to operate atomically with respect to the CloseAuction step. Thus, the more detailed model provided here leads to improved analysis capabilities.

Modeling a global property like AuctionClosed and exporting more events from the composite machines is likely to make the model grow. On the other hand, the synchronization that results by modeling the AuctionClosed variable may reduce the size of the model as it makes certain interleavings impossible. The overall impact of modeling the AuctionClosed global variable, exposing the necessary events, composing in the property, and modifying the process to include the extra prerequisite on UpdateBestBid is shown in Table 2. Comparing this with the previous results, we see that the pre-minimization sizes of the steps from UpdateBestBid up the step hierarchy to OpenCryAuction are roughly doubled to account for the extra events that are now being included in those models. Similarly, the post-minimization sizes of the models for the steps from UpdateBestBid through AcceptBidsFromBidders are also doubled. The interesting point, though, is that once we reach the step where the property is proven, it is no longer necessary to export these extra events. As a result, the minimized sizes of the root step are identical in the two cases. If the property could be proven lower in the step hierarchy, it would have impacted fewer step models.

6. FUTURE WORK

The goal of this work is to be able to prove process-specific properties. Currently, doing so requires some hand manipulation of the process to limit recursion and some hand manipulation of the generated model to define properties, modify parameters, rename events, and expose events. We are currently working on a tool to assist an analyst in these tasks by helping the analyst find the recursive steps and modify the process to bound recursion as the analyst sees fit, including modeling the condition that causes the recursion to stop. A further step would be to help the analyst in defining properties at the level of the Little-JIL process so that those properties and associated changes to the model could be made automatically.

7. CONCLUSIONS

Model checking operates on a model that abstracts away some details of the code. One of the challenges in model checking is to create a model that is sufficiently abstract to remain computationally feasible while sufficiently detailed to capture the semantics of the application that are being verified. Little-JIL provides such an abstraction for applications that involve the coordination of multiple agents by focusing on the coordination issues and abstracting away the details of agent behavior. Furthermore the style of implementation of the Little-JIL interpreter has given us the ability to model Little-JIL processes in terms of their execution behavior, not just in terms of the intended semantics of the Little-JIL constructs.

Our results in uncovering errors in the implementation of the interpreter suggest that this style of implementation, with state machines explicitly embedded in the source code, facilitates the development of an accurate model. Adopting this implementation style in other applications is likely to also facilitate model checking in those applications. In addition, the hierarchical nature of Little-JIL steps made the models more tractable in size as the model could be minimized at each level of the hierarchy, suggesting that workflow and process languages with hierarchical representations may be more amenable to analysis than the more common representations based on data flow.

Acknowledgements

Little-JIL was designed and developed by members of the Laboratory for Advanced Software Engineering Research at the University of Massachusetts, Amherst under the guidance of Lee Osterweil. The state machine design of the interpreter was done chiefly by Sandy Wise, Eric McCall, and Aaron Cass. Aaron was responsible for the state machine based implementation of the Little-JIL interpreter using SMC. The auction process was developed by Aaron, Sandy and Hyungwon Lee and previously appeared in an ISSTA paper. Jamieson Cobleigh has given me helpful tips on how to use LTSA effectively. Finally, I would like to thank Rick Lerner, Jamie and Aaron for their comments on an earlier draft.

This material is based upon work supported by the National Science Foundation under Grant Numbers CCR-9988254, CCR-0204321 and CCR-0205575. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[2] S. C. Cheung and J. Kramer. Checking safety properties using compositional reachability analysis. *ACM Trans. on Soft. Eng. and Methodology*, 8(1):49–78, Jan. 1999.

[3] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. Verifying properties of process definitions. In *Proc. of the Intl. Symp. on Software Testing and Analysis (ISSTA)*, pages 96–101, Portland, Oregon, August 2000.

[4] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. of the 22nd Intl. Conf. on Soft. Eng.*, pages 439–448, Limerick, Ireland, June 2000.

[5] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services, version

	Before minimization		After minimization		Composition	Minimization
Step	States	Transitions	States	Transitions	time (sec)	time (sec)
OpenCryAuction	105	145	22	28	0.3	0.05
CloseAuction	92	175	14	26	0.3	0.02
AcceptBidsFromBidder	286	830	38	97	0.3	0.1
AcceptOneBid	384	1130	59	147	0.3	0.1
SubmitBid	126	221	17	31	0.3	0.03
Update Best Bid	254	744	41	109	0.3	0.1
Total					1.8	0.4

Table 2: Sizes of the Models of the Steps in the Modified OpenCryAuction Process Including a Process-Specific Property and Time to Create Them

1.0. Technical report, BPMI.org, July 2002. available at http://www.bpmi.org/bpml-spec.esp.

[6] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent java programs. *Software - Practice and Experience*, July 1999.

[7] 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 Soft. Eng.*, pages 62–75, December 1994.

[8] R. Eshuis and R. Wieringa. Verification support for workflow design with UML activity graphs. In *Proc. of the 24th Intl. Conf. on Soft. Eng*, pages 166–176, Orlando, Florida, May 2002.

[9] H. Foster, S. Uchitel, J. Magee, and J. Kramer. Model-based verification of web service compositions. In *Proc. of the 18th IEEE Intl. Conf. on Automated Soft. Eng. (ASE)*, Montreal, 2003.

[10] G. Holzmann. A practical method for verifying event-driven software. In *Proc. of the 21st Intl. Conf. on Soft. Eng. (ICSE'99)*, pages 597–607, May 1999.

[11] C. Karamanolis, D. Giannakopoulou, J. Magee, and S. Wheater. Model checking of workflow schemas. In *Proc. of the 4th Intl. Enterprise Dist. Object Computing Conf. (EDOC'00)*, Makuhari, Japan, Sept 2000.

[12] M. Koshkina and F. van Breugel. Verification of business processes for Web services. Technical Report CS-2003-11, York University Department of Computer Science, Oct. 2003.

[13] F. Leymann. Web services flow language (wsfl 1.0). Technical report, IBM, 2001. available at http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

[14] J. Magee and J. Kramer. *Concurrency: State Models and Java Programs.* John Wiley & Sons, 1999.

[15] S. Nakajima. Model-checking verification for reliable web service. In *Proc. of the OOPSLA 2002 Workshop on Object-Oriented Web Services*, Seattle, Nov. 2002.

[16] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proc. of the 21st Intl. Conf. on Soft. Eng.*, pages 399–410, Los Angeles, 1999.

[17] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proc. of the 4th Joint Meeting of the European Soft. Eng. Con. and ACM SIGSOFT Symposium on the Foundations of Soft. Eng. (ESEC/FSE 2003)*, March 2003.

[18] R. W. S. Rodrigues. Formalising UML activity diagrams using finite state processes. In *Proc. of the 3rd Intl. Conf. on the Unified Modeling Language*, York, UK, Oct. 2000.

[19] S. D. Stoller. Model-checking multi-threaded distributed java programs. In *Proc. of the 7th Intl. SPIN Workshop*, pages 224–244, 2000.

[20] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of the Intl. Conf. on Automated Soft. Eng.*, Sept 2000.

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

[22] A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and Stanley M. Sutton, Jr.. Using Little-JIL to coordinate agents in software engineering. In *Proc. of the Automated Soft. Eng. Conf. (ASE 2000)*, pages 155–164, Grenoble, France, September 2000.

[23] W. J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proc. of the Symp. on Testing, Analysis, and Verification (TAV)*, pages 49–59, October 1991.