Developing Discrete Event Simulations From Rigorous Process Definitions

Mohammad S. Raunak[1], **Leon J. Osterweil, Alexander Wise**
Department of Computer Science
University of Massachusetts Amherst
{raunak, ljo, wise}@cs.umass.edu

Keywords: Executable process models, Resource Management, Complex and Dynamic systems, Human-centric processes, Health care simulation.

Abstract

A process modeling language that is easy to learn and use while grounded in rigorous semantics to facilitate execution and simulation has always been a big challenge for researchers. In this paper, we describe Little-JIL, a visual process modeling language and its runtime-infrastructure, Juliette, that is capable of driving a process with the help of participating human or computing agents. We then introduce JSim, a discrete-event simulation environment with flexible artifact management and intricate resource management capability built on top of the Little-JIL and Juliette framework. The factored architecture and rich modeling mechanism for both the process description as well as the resource specification has allowed us to simulate some dynamic and complex real-life systems in details.

1. MOTIVATION

Sufficiently precise and completely defined processes can be used as the basis for efforts at process improvement, triggered by the identification and subsequent removal of process defects using different types of static analysis techniques such as finite state verification [4], or fault-tree analysis [5]. Such static analysis do not require the execution of the process, but then also do not benefit from various advantages that would be offered by the executability of the defined process. The use of an executable process definition, for example, creates the possibility that the iteratively improved process could, at each iteration, be executed to provide increasingly effective and satisfactory results. This, in essence, boils down to the use of dynamic analysis approaches to support iterative improvement of processes as complements to the static analyses used to identify potential process defects. In this paper we describe such an effort, namely the use of a sufficiently complete, precise, and executable process definition as the basis for discrete event simulation. In this paper we show how we have been able to generate discrete event simulations from an executable process definition language. One important advantage of this approach is that we can use the same process models for developing simulations that have already been used for other analyses purposes. This reduces the considerable cost of developing precise process models by amortizing it over various analyses. It also ensures that different analyses were aimed at studying and improving the same process.

Another important contribution of this paper is presenting the importance of careful resource (including actors) modeling and management for proper process simulation. Executable modeling languages for business processes such as WSBPEL [8] and its extension to include humans [9] have lacked detailed definition of resources [11]. Similarly, although there have been a number of discrete event simulation efforts for modeling and simulating software processes [17], very few of these simulation models paid attention to modeling *resources* as well as *process context* adequately. This paper describes how our focus on resource modeling and management has resulted in sufficient details of particularly complex and dynamic human centric processes where process context may dictate resource utilization.

2. APPROACH

Our approach in developing a process simulator starts with the use a complete, yet nicely factored process definition language, whose semantics is rigorously defined to enable process execution. In our view, an executable process definition framework needs to have a coordination aspect focusing on activities and the flow of control, a resource aspect focusing on actors and tools required to carry out the activities and the runtime binding of actors and other resources to activities, and an artifact aspect focusing on data that are produced, consumed and shared by actors in performing their assigned work. A process interpreter can then be developed to take the static coordination specification and provide communication and synchronization mechanism with modules defining the artifact model, providing resource-management services, and the actors responsible for carrying out the activities. Given such a process definition and interpretation framework that provides the means for coordination amongst participating actors through maintaining the control and data flow and thus enabling execution of the process, one can extend the architecture to replace the real-life actors with simulated actor behaviors to create a flexible simulation service. This is precisely what this paper describes as the methodology we have used. We have developed a process simulation infrastructure

[1]The author is currently a visiting Assistant Professor at Loyola University Maryland.

called JSim leveraging the flexible architecture of a process execution framework.

2.1. Little-JIL: A Rigorous and Executable Process Language

Little-JIL, the modeling language used as the basis for our discrete-event simulation infrastructure, is a process definition language [3, 14] that, along with Juliette, its interpretation framework, supports specification, execution, and analysis of processes that are performed by sets of agents (actors) that may be humans, software, or hardware devices. A Little-JIL process definition is composed of three orthogonal aspects: a visual activity coordination specification, a collection of artifacts with an artifact-flow specification, and a collection of resources.

The activity coordination specification is a hierarchical structure of *steps* each of which is an abstraction of an activity. Child steps are connected to their parent by edges that represent both *control flow* and *artifact flow*. Each step contains a specification of the type of agent resource needed to perform the task associated with that step. Thus, for example, in a hospital patient care process, the agents would be doctors, nurses, registration software etc. Specific agent instances are assigned to steps at runtime by a resource manager (all agents are considered to be resources). The collection of steps assigned to an agent defines the interface that the agent must satisfy to participate in the process. Note that the coordination specification includes a description of the external view and observable behavior of such agent resources. But a specification of how the agent resources themselves perform their tasks (their internal behaviors) is NOT part of the coordination specification. These behaviors are defined in a separate specification component.

More formally, the activity coordination specification of a Little-JIL process definition P is a tree structure (ST, rt, E, R, F) where:

- ST is a set of steps, each of which is an abstract specification of an activity that can be instantiated multiple times in an actual execution or simulation of a process.
- A root step, $rt \in ST$, that defines the beginning or entry point for P.
- E, a set of edges each of which is an ordered pair of steps $(st_p, st_c) \in E$ connecting st_p and st_c in a parent-child relationship.
- R is a set of resource specifications associated with each step st.
- F is a set of artifact specifications associated with step declarations and edges connecting the steps.

We now briefly introduce the syntax and semantics of a Little-JIL process definition. A *step*, the primary building block of a Little-JIL process, has a number of badges associated with it, which provides its semantic. The *interface badge* is a circle on the top of the step name that connects a step to its parent. The interface badge holds information about the artifacts used in and produced by the step as well as requirements for resources needed for the execution of the step. The step's *execution agent* requirement is specified here as well. Below the circle is the step name. A step may also include pre-requisite and/or post-requisite badges, which are representations of simple predicates or entire step structures that are to be executed before and/or after (respectively) the step. Inside the central black box of the step structure, are three more badges. On the left is the control flow badge, which specifies the order in which the step's children are to be executed. Based on the control-flow, Little-JIL has four different step kinds, namely *sequential*, *parallel*, *try* and *choice*. Children of a *sequential* step, as seen in most non-leaf steps in figure 1, are executed one after another from left to right. Children of a *parallel* step can be executed in any order, including in parallel, depending on when the agents actually pick up, and begin execution of, the work assigned in those steps. A detailed discussion of step kinds is available at [14].

On the right of the step bar is an X sign, which represents the exception handler capabilities of the step. Attached to this badge by red-colored exception edges are any and all handlers defined to deal with exceptions that may occur in any of the descendants of this step. Each handler may itself be a step, and is annotated to indicate the type of exception that it handles. Here too, artifact flow between the parent and the exception handler step is represented by annotations on the edge connecting them. When an exception is thrown by a step, it is passed up the tree until a matching handler is found. Little-JIL also defines four different continuation semantic after an exception has been handled [14]. In the middle of the step bar is a *lightning sign*, which represents the message handling capabilities of the step. Attached to this badge by message handling edges (also known as reaction handling edges) are any and all handlers defined to deal with messages that may emanate from any step in the process definition or even from outside the process. The message handling capability is quite similar to the exception handling capability, but, while exception handlers respond only to exceptions thrown from within their substep structure (a scoped capability), message handlers can respond to message thrown from anywhere (an unscoped capability). If there are no child steps, message handlers, or exception handlers, the corresponding badges are not depicted in the step bar.

2.2. A Little-JIL Process Example

To illustrate how Little-JIL can be used to define a process, we present the 'EDCare' process, a simplified model of how care is provided to hospital Emergency Department (ED) patients, shown in figure 1. EDRoot, the root step of the EDCare process, has a sub-process structure defined as a reaction

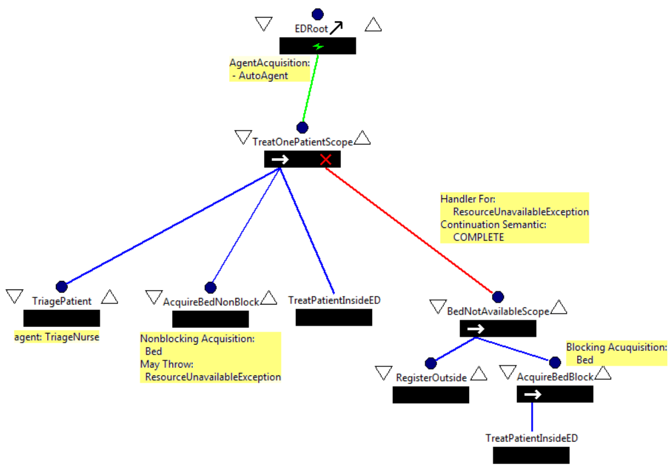

Figure 1. The root diagram of 'EDCare' process

handling scope. When the root step receives a message of type PatientArrivalMessage (specified inside EDRoot step's interface badge), the process sub-tree TreatOnePatientScope that defines how patient care is provided, is instantiated. Thus each PatientArrivalMessage results in a potentially parallel instantiation of the process structure rooted at TreatOnePatientScope. For each patient, the process begins with a triage nurse performing triage on the patient, as defined by the TriagePatient step. The required resource characteristics for the agent that can perform this task is defined as part of the step, but, is not visible in the figure to reduce visual clutter. One outcome of executing this step is the assignment of an acuity level for the patient. In a simplified nominal version of the ED process, a patient next goes for Registration where insurance and other information is taken from the patient and an Id-band is placed on the patient. The patient next waits to be taken into the treatment area of the ED (the mainED) depending on the availability of bed. However, figure 1 describes a variation of the simple patient care process. This was modeled in order to study the impact of a policy change where a patient can be placed inside the mainED without waiting to undergo the regular registration activity. Inside the mainED, (defined in the elaboration of the reference step TreatPatientInsideED) if a patient has not yet gone through registration, a shortened registration called *quick-registration* is performed first, and then the rest of the registration (e.g. collecting insurance information) is completed in parallel with the treatment process. The process flow, in this case, takes a patient through the TriagePatient step, and then immediately tries to acquire a bed through the AcquireBedNonblock step. One of the required resources for this step is the bed. If the bed acquisition is successful, process flow continues to the sub-tree rooted at TreatPatientInsideED. In case AcquireBedNonblock fails to acquire a bed an exception of type ResourceUnavailableException is thrown and propagated up to the TreatOnePatientScope step, where a separate process (BedNotAvailableScope) is defined to handle the exception. The handler process, starts by performing the step RegisterOutside and then tries to acquire a bed with a blocking request call. TreatPatientInsideED is a reference step that refers to the root of another Little-JIL diagram that elaborates the treatment process inside the mainED. There is a parameter named *patientInfo* that is instantiated with the information about each patient at TreatOnePatientScope and is flowed through the process from step to step as an artifact. Attributes of this artifact can be used as predicates associated with the pre-requisites and post-requisites of the steps that this artifact flows through. Note that the yellow post-it notes in figure 1 are comments and not part of the formal process definition.

3. LITTLE-JIL RUNTIME ENVIRONMENT

Juliette, the Little-JIL runtime environment, uses *agendas* (to-do lists) to coordinate agents $\{A_1, A_2, A_3, \cdots A_n\}$ that are specified to execute a Little-JIL process definition. Each agent A_i has associated with it an agenda N_{A_i}, which is the public interface that allows activities to be assigned to that agent. Thus, at any time during the execution of a process, $N_{A_k} = \{N^1_{A_k}, N^2_{A_k}, \cdots N^n_{A_k}\}$ is a potentially empty set of agenda items, where an agenda item $N^i_{A_k}$ is an instance of an activity assigned to agent A_k. Note that at runtime, a step st_m, which is a type level specification of an activity, may get instantiated many times. Let $ST_m = \{si^1_m, si^2_m, \cdots si^n_m\}$ denote the set of all instances associated with step $st_m \in ST$ in an execution of a process P. Since instances of steps are carried out by agents, a step instance $si^j_m \in ST_m$, when bound to an agent A_k becomes an agenda item $N^i_{A_k}$. In Little-JIL runtime environment, this is done by using a common abstraction for both the step instance si^j_m and its corresponding agenda item, such as, $N^i_{A_k}$.

Figure 2 shows the component structure of Little-JIL runtime environment. The solid boxes represent main components of the system. A solid directional line defines a call from one component to another (the *uses* relationship). When two components communicate with each other asynchronously, it is shown with bi-directional lines connecting the components. The process specification has been shown as a broken line box with its three components inside it.

The lifecycle of a step's execution is defined through a finite state machine that defines how execution moves through the key states (*initial*, *posted*, *started*, *completed*, or *terminated*) and transitional phases (*elaboration*, *starting*, *execution*, *finishing*). The lifecycle of a step instance si^j_c is as follows.

- *Elaboration phase*: In this phase, the Step Interpreter copies artifacts from step instance si^j_p based on the parameter flow annotations between st_p (parent) and st_c (child). The Resource Manager is then sent the agent

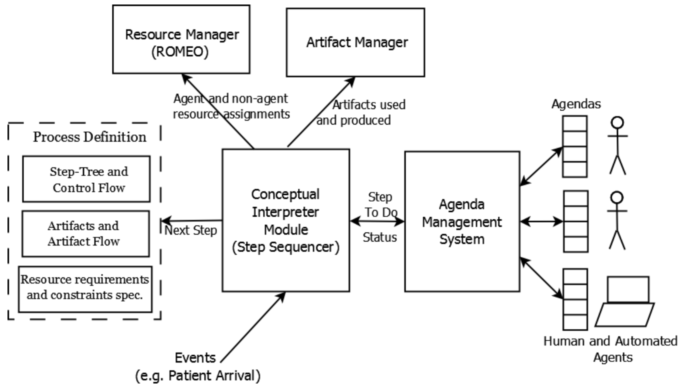
Resource Manager (ROMEO)
Artifact Manager
Agent and non-agent resource assignments
Artifacts used and produced
Agendas
Process Definition
Step-Tree and Control Flow
Artifacts and Artifact Flow
Resource requirements and constraints spec.
Next Step
Conceptual Interpreter Module (Step Sequencer)
Step To Do
Status
Agenda Management System
Human and Automated Agents
Events (e.g. Patient Arrival)

Figure 2. Little-JIL Runtime Environment

resource requirement associated with the definition of the step being elaborated, i.e., st_c. The Resource Manager makes a decision on an agent A_k that matches the requirements. With a message back from the Resource Manager, the Step Interpreter assigns the step instance si_c^j to agent A_k by placing an agenda item $N_{A_k}^n$ in A_k's agenda. This takes si_c^j from a *initial* state to *posted* state.

- *Starting phase*: The agent A_k chooses to start the activity si_c^j at some point t, which chages si_c^j to be in *started* state.
- *Execution phase*: The execution phase defines what happens while si_c^j is in *started* phase. If si_c^j is a leaf step, A_k will complete the activity potentially making changes to the artifacts that were passed to it and then notifies the Step Interpreter that it has completed the work. If si_c^j, on the other hand, is a non-leaf step, the Step Interpreter will drive the execution of each of its sub-steps following the sequencing specification associated with static definition of the step st_c.
- *Finishing phase*: As part of the finishing state of a step instance's life cycle, the Step Interpreter copies out the values of the parameters that have been specified to be returned from si_c^j to its the parent step instance si_p^j. The Resource Manager is then asked to release the agent A_k and other resources used in si_c^j. During this phase si_c^j transitions from *started* state to *completed* state or to the *terminated* state in case si_c^j could not complete normally for some reason.

Thus, the execution of a process is a resultant behavior that emerges from the interconnected execution of step instances. This high level description leaves out a number of Little-JIL language features and details of the Juliette process execution environment. More complete discussions about the language and the interpretation are available at [3, 16, 14]. One important thing to note here is the separation of concerns enforced by the runtime architecture. The step interpreter interacts with other components at specific points in the life cycle of a step and this interaction is governed by well-defined APIs that restricts the type of information communicated. Artifact types and their flow behavior are defined within the Artifact Manager. The decision to select an agent and to couple its agenda to a step instance is done by the Resource Manager. The agents participate in the process by fulfilling the contract specified by their interaction API in the AMS. With this architecture, the interpreter can assure that the control and artifact flow semantics of Little-JIL are enforced without needing to know anything about the resource management policies or artifact type structures. However, the participating agents need to register with the Agenda Management System making their *agenda* public. A participating agent is also expected to listen for *agenda items* placed in its *agenda* and fulfill its contract by notifying the corresponding Step Interpreter when it has completed its work or if it decides to terminate the work failing to complete it.

4. THE JSIM ARCHITECTURE

Section 3 has described an execution framework capable of driving a process definition with the participation of agents and through the interaction of different components such as Step Interpreters, Resource Manager and Artifact Manager. In a simulation environment, we are not going to have live agents (humans or computer programs) interacting with the interpreters. Thus an essential part of developing a discrete event process simulator would require replacing the interactions with agents by a component capable of producing synthetic, yet flexible, agent behavior. The following diagram describes the JSim component structure.

In figure 3, the process definition represents both the static step definitions as well as runtime step instances. Each Step Instance object has a Step Interpreter associated with it that maintains and drives the life cycle of the Step Instance according to a finite state machine defined by the type of the Little-JIL step [3]. These finite state machines are already specified as part of Little-JIL semantics and thus do not require any additional programming. The Simulator Agenda Management System (SAMS), a little modified implementation of the original AMS used for process execution, is at the heart of maintaining and facilitating all the communications amongst the Step Interpreters and other modules in the runtime system, such as the Resource Manager and the Artifact Manager. As noted earlier, the important difference here from the generic runtime environment of Little-JIL is the absent of real agents, which were shown in figure 2 as execution clients. One of the components that would now become the client of SAMS service is the JSim Agent Behavior Specification (JABS) module [15], which holds specification of agents' behaviors. This module answers queries as to what an agent would do when it is given a step instance to perform. A Step Instance object, with the help of its associated Step Interpreter object communicates with other Step Instance objects,

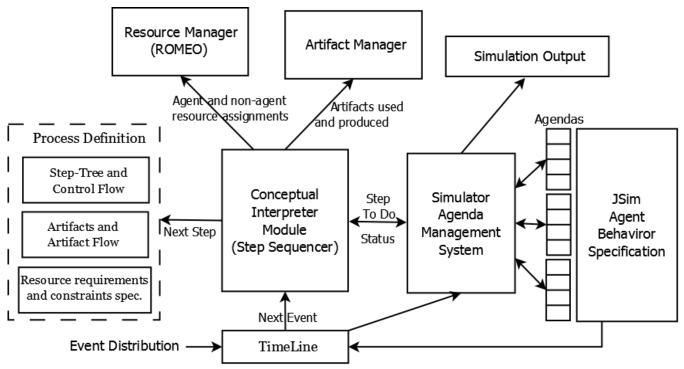
Resource Manager (ROMEO)
Artifact Manager
Simulation Output
Agent and non-agent resource assignments
Artifacts used and produced
Agendas
Process Definition
Step-Tree and Control Flow
Artifacts and Artifact Flow
Resource requirements and constraints spec.
Next Step
Conceptual Interpreter Module (Step Sequencer)
Step To Do
Status
Simulator Agenda Management System
JSim Agent Behaviror Specification
Next Event
Event Distribution
TimeLine

Figure 3. Subsystem view of JSim process simulator

agents, and other components of the runtime environment using the Simulator AMS service. It should be noted here that the communications between Step Instances and agents are asynchronous. In the simulation environment, with the absence of real agents, a mechanism is needed to notify a Step Instance that is waiting to hear back from an agent. This is accomplished with the Timeline module and the Simulator Agenda Management System.

The Timeline module holds all the events generated by simulated agents as specified in the Agent Behavior Specification component as well as events corresponding to some global messages. With the above components in place, a simulation driver is used to serially read events from the Timeline module and to deliver them to appropriate Step Instance objects. The event deliveries result in state changes in the Step Instance objects, which may generate more events to be created and placed in the Timeline. The Simulation driver waits until no more events can be generated as a result of an event delivery. Once all subsequent events that resulted from an event delivery to the SAMS have been placed in the timeline, the simulation driver goes on to pick the next event and delivers it. The simulation stops when there is no more event to be delivered.

In practice, the process definition component may often be reused from earlier construction of the process for different static analysis purposes. The resource requirement specification, and resource related constraints are usually added to the process and resource definition for simulating specific scenarios. If the process definition has been used before for execution, then there is almost no change required in resource or artifact models. It is the JSim agent behavior and the initial distribution of events in the timeline that requires programmatic specification to setup a specific simulation. We have developed tools to partially automate the process of generating agent behavior as well as initial event distribution.

4.1. ROMEO Resource Management Service

One of the primary purposes of studying simulations of processes is to look at the resource contention and utilization and its impact on some measurement of production or service time under different contextual settings including variations in service request load and available resources. In processes such as software development or patient care services in an ED, the resources of interest mainly includes human actors in that domain such as software designers, developers or doctors, nurses, clerks etc. In a patient care process, there are also non-agent resources such as the bed, medical equipment, medicine etc., which are of crucial importance. These resources are quite diverse in their characteristics. Some can participate as actors in a process (such as the humans) while others can not (e.g. bed in an ED), some are reusable (e.g. bed) while others are consumable (e.g. medicine). More intriguingly, some of these resources, especially the agent resources, may provide capabilities only under some circumstances and not under others. We note, for example, a physician assistant in an ED may perform an activity such as writing an order (i.e. prescribing medication) for a patient with chest pain in an extraordinary situation, whereas this is a task that would only be performed by a doctor under ordinary circumstances. It is thus imperative to have a flexible resource management service that can model such diverse kinds of resources with often complex and dynamic characteristics adequately.

As an essential support service for both Little-JIL process execution as well as JSim simulation infrastructure, we have developed a flexible resource management service named ROMEO. During the execution or simulation of a Little-JIL process, when activities become ready to get started, ROMEO is sent these required resource specifications as requests for resource acquisitions. The ROMEO resource manager, in turn, performs a search through its repository of resources, identifies resources capable of fulfilling the required specification and chooses one of those resources based on some suitability or utility metric. The resource request specifications in a Little-JIL step may also carry indication as to whether the process will wait until resources become available (blocking request) or whether it wants to be notified immediately by the Resource Manager in case a resource was not readily available (non-blocking request).

When ROMEO receives requests for agents and other resources that can not be immediately fulfilled, it puts them in a queue of pending requests. As the simulation progresses, when resources become free as a result of completion of activities that had started earlier, the resource manager picks up requests from the pending queue and fulfills them. One of the important features of ROMEO is its rich modeling mechanism of both agent and non-agent resources, the request language to specify resource requirement, and allocation constraint that can be specified as part of the Little-JIL process definition or the resource model in ROMEO. The key idea behind ROMEO's modeling of resources is that it considers a resource, especially the agent resource, as a provided of a set of *capabilities*. For example, in the ED, a nurse resource can provide such capabilities as *triage*, *administer-medication*, *vital-check* etc. However, each capability has associated with it a guard function that defines under what circumstances can the resource provide that capability. This provides us with considerable flexibility in modeling dynamism in a simulated process. Space limitation does not allow us to write about the details of ROMEO's architecture, its resource modeling mechanism, the request language that it supports and its allocation strategies here. We have extensively written about our resource management capabilities in [10].

4.2. JSim Agent Behavior Specification

To allow us to specify simulated agent behaviors in a flexible way, we have developed an XML based rule language, named the JSim Agent Behavior Specification (JABS) language [15]. In a Little-JIL process execution environment, once an agent is assigned some activity, it participates in the running process in two ways. First, it notifies the interpretation module when an activity is started by the agent and when it is completed. Second, the agent may use, modify or produce artifacts as part of performing the activity. These artifacts are passed into and out of a Little-JIL step using parameters that define the flow of the artifacts between steps.

JABS allows us to specify the behavior of an agent or a set of agents based on the static name of a Little-JIL step or based on the name of the agent itself. When a rule is defined in terms of a step name, the behavior of any agent assigned to perform an instance of that step is deduced from this rule. Each rule in JABS language is composed of a matching part and a set of action elements. The matching part holds the name of a step or an agent. The action elements specify information such as the relative start time and end time of a step with respect to the current simulation clock time when an instance of this rule is invoked. The action elements may also specify setting of any field of an artifact or the entire artifact that is bound to the step where this rule is being applied. While specifying timing information an agent would take to perform a step, it is important to provide some statistical distribution. JABS allows us to specify uniform, triangular and Gaussian distribution as the time it takes to perform any step. It is possible to have multiple rules that match a particular simulation context of a step execution. In this case, JABS performs a top down search in the rule base and selects the first matching rule and applies it. The JABS module also allows for arbitrary nesting of the agent behavior rules.

4.3. Generating Simulation Output

The raw output of JSim are the lifecycle events and their occurrence time according to the simulation timeline. The following list describes the major events that are logged by JSim.

1. A step instance si_c^j is ready to be *posted* at time t_1 based on the control flow of the process definition.
2. The Resource Manager assigns an agent A_k to carry out step instance si_c^j at time t_2.
3. Agent A_k decides to start executing si_c^j at time t_3.
4. Agent A_k completes si_c^j at time t_4.

Using the data captured in the trace information, we have been able to compute a wide number of statistics of interest for many simulated processes. Most of the statistics we have generated fell under the following two categories:

- *Time information*: we can compute the time taken for executing any leaf or non-leaf step. This translates to time information regarding activities at different levels of granularity captured in the process definition. We can also compute time information specific to any particular agent.
- *Resource Utilization*: The trace provides data specifying when tasks were assigned to agents, when they started working on a task, and when did they complete a particular task. From these time information, we can compute utilization level of the agents and other resources.

4.4. Validation of JSim

Extensive efforts were made to assure that simulation output produced by JSim was consistent with its specifications [10]. Some of these efforts amounted simply to careful visual scrutiny of the resource assignment decisions as well as ensuring that starting and completion of the steps are happening according to the simulation specification. Both JSim developers and our ED domain expert spent considerable amounts of time looking at simulation results to verify JSim's simulation traces. We also compared the results obtained from JSim to those predicted by Little's Law [2, 10], a well-known "rule of thumb" used to estimate and validate the overall behaviors of queue-based simulations.

To gain more confidence in the inner workings of the JSim infrastructure and simulation results produced by it, we decided to compare our results with a well established commercial discrete event simulation product: Arena [6, 1]. Arena is an object-based, hierarchical modeling and simulation tool that has been used in a wide range of simulation applications. Of particular relevance to our studies is the fact that many ED simulation studies have used Arena as their modeling tool.

For purposes of comparison, we modeled a simple ED process using both JSim and Arena and ran simulations with patients arriving at a fixed rate. The process definition we used for this study is similar to the one showed in figure 1, albeit a little more simplified. The task times were kept fixed and only one type of resource was varied, namely the bed resource. We plotted both how long the simulation ran in total time as well as the average length of stay (LOS) for each patient. We went through a number of iterations of simulations and inspections to ensure that details of both the Arena and the JSim models were describing the exact same process. The output produced by the two simulation engines were same [10].

5. EXPERIENCE AND CASE STUDIES

We have used Little-JIL, ROMEO and JSim to simulate a number of different processes related to how patient care is provided in a hospital ED. With Little-JIL, JSim and ROMEO, we have been able to simulate some complex and dynamic scenarios related to ED. While designing case studies, we investigated the following two attributes of our JSim simulation infrastructure:

- How capable is our modeling mechanism to capture some intricate domain policies in such dynamic processes in hospital EDs.
- How easy or difficult is it for a novice modeler to model new complex simulation using JSim.

We have found that the JSim architecture affords expedited access to each of the key architectural components, and many different points at which the parameters used to configure these components can be tuned and adjusted quickly. This has been useful in facilitating the tuning and setup of specific simulations or sets of simulations. We have also found that the non-technical domain experts (in this case, doctors and nurses) quickly picked up the Little-JIL notation and were able to follow and suggest detailed specification in the process definition.

We now present an example of the kinds of experiments we have performed with JSim. Earlier in the paper, while introducing the Little-JIL notation, we described the 'ED-Care' process. This process was developed with by extending a nominal and simpler ED process that we used for initial validation purposes. Here we have added different types of resource acquisitions (both blocking and non-blocking). As noted earlier, there is a detailed part of the process that is rooted in the TreatPatientInsideED step, which has not been shown in figure 1.

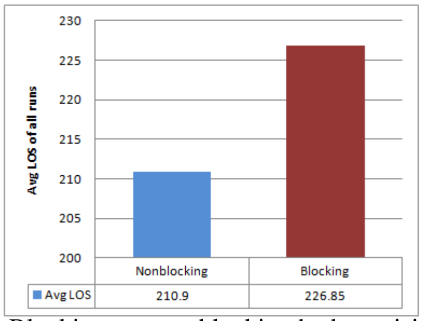

Figure 4. Blocking vs. non-blocking bed acquisition

For this set of simulation runs, the patient arrivals were generated using a *Poisson* distribution with mean inter-arrival time of six (6), which translates into roughly ten (10) patients per hour. The execution times of the steps were specified using a triangular distribution. We looked at the average of all the average LOS measures from the simulation runs. Figure 4 summarizes the output of these simulations. As intuitively expected, the situation where a patient is immediately placed inside the ED when a bed is available results in improved patient flow. We were also interested in observing how easy or difficult it is to set up our simulation and resource management infrastructure for such an experiment. So, we assigned the task to a relative novice of Little-JIL and JSim, a graduate student with less than 2 months experience in our lab, and timed him in doing the simulation setup. The process augmentation required about 2 hours and 24 minutes. Once we had the process model elaborated, switching from a non-blocking request scenario to the blocking request scenario to setup this particular experiment required less then twenty (20) minutes to complete. Our other experiments with a novice user have also validated JSim's ability of expressing complex scenarios easily and quickly. In one of the ED simulations, we modeled a scenario where a triage-nurse could perform some of the discharge-related paper-works that is usually performed only by a regular nurse. But we allowed the substitution only when the ED was overcrowded. Details of this and all our other experiments can be found in [10].

6. RELATED WORK

Different workflow and process modeling languages have been used by researchers over the past decade to simulate software development and other human-centric processes [17]. The capabilities in these languages are usually restrictive, however, especially when it comes to resource specification and management. BPEL4WS [8] and BPEL4-People [9] haven been popular for modeling business processes for simulation and analysis. These languages have also been shown to lack in terms of capabilities for modeling resources and process context details [11].

Schriber [12] provides a nice explanation of the generic structures of discrete-event simulation software with an analysis of four commercial tools, namely SIMAN, ProModel, GPSS/H, AutoMod and SLX [7]. Many of these commercial simulation tools presented in [12] as well as a number of new ones such as Arena [1], AnyLogic [13] and others have nice user interfaces, but often are not as programmable as our JSim simulation infrastructure. In particular, the resource modeling and managing flexibility that we can deliver using JSim seems to be very difficult, if not impossible, with many of these commercial tools. The rich exception management mechanism of Little-JIL and contextual assignment of resources to activities are features that seem to set our infrastructure apart.

7. CONCLUDING REMARKS

We believe there is considerable value in creating a discrete-event simulation infrastructure based on rigorous and executable process definitions that is accessible to non-technical domain experts. The formal semantics of such process modeling languages would also allow one to perform different kinds of static analysis of processes [4, 5]. In this paper we have presented such a process definition language - Little-JIL, its runtime environment - Juliette, and a discrete event simulation framework that we have developed on top of them called JSim. One important component of JSim is its rich resource management component, ROMEO, which is capable of modeling dynamic agent behavior that is dependent on the state of the system. Using this infrastructure, we have successfully simulated some intricate variations of how patient care is provided in a hospital ED [10]. We have also performed case studies to evaluate the versatility and usability of our simulation framework.

Acknowledgement

The authors gratefully acknowledge the important contributions made to this work by Guillaume Viguier-Just and Tiffany Chao, who provided support for the construction of case studies and other evaluative activities. Dr. Philip L. Henneman, MD, Prof. Hari Balasubramanian, Prof. Lori Clarke, and Prof. George Avrunin provided a great deal of help and support through their continuing interaction about this work.

This work was supported by the US National Science Foundation under Award Nos. CCR-0427071, CCR-0204321 and CCR-0205575. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of The National Science Foundation, or the U.S. Government.

Biography

Dr. Mohammad S. Raunak is currently a Visiting Assistant Professor at Loyola University Maryland. His research interests include process simulation and analysis of complex systems with a special focus on resource management. Prof. Leon J. Osterweil is an ACM fellow and a professor of University of Massachusetts Amherst. Mr. Alexander Wise is a senior technical staff at Laboratory for Advanced Software Engineering Research (LASER) at University of Massachusetts Amherst.

REFERENCES

[1] V. Bapat and D. T. Sturrock. The arena product family: enterprise modeling solutions: the arena product family: enterprise modeling solutions. In *Proceedings of the 35th conference on Winter simulation*, pages 210–217, 2003.

[2] D. Bertsimas and D. Nakazato. The distributional little's law and its applications. *Operations Research*, 43:298–310, 1995.

[3] A. G. Cass, B. S. Lerner, S. M. Sutton, E. K. McCall, A. E. Wise, and L. J. Osterweil. Little-jil/juliette: a process definition language and interpreter. In *International Conference on Software Engineering (ICSE)*, pages 754–757, 2000.

[4] B. Chen, G. Avrunin, L. Clarke, and L. Osterweil. Automatic fault tree derivation from little-jil process definitions. In Q. Wang, D. Pfahl, D. Raffo, and P. Wernick, editors, *Software Process Change*, volume 3966 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

[5] B. Chen, G. S. Avrunin, E. A. Henneman, L. A. Clarke, L. J. Osterweil, and P. L. Henneman. Analyzing medical processes. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, 2008.

[6] J. E. Hammann and N. A. Markovitch. Introduction to arena. In *WSC '95: Proceedings of the conference on Witer Simulation*, San Diego, CA, USA, 1995. Society for Computer Simulation International.

[7] J. O. Henriksen. An introduction to slx. In *Proceedings of the 29th conference on Winter simulation*, WSC '97, pages 559–566, Washington, DC, USA, 1997. IEEE Computer Society.

[8] D. Jordan and J. Evdemon. Web services business process execution language version 2.0; oasis standard. Technical report, April 2007.

[9] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. Riegen, P. Schmidt, and I. Trickovic. WS-BPEL extension for people. Technical report, IBM, 2007.

[10] M. S. Raunak. *Resource Management in Complex Dynamic Environments*. PhD thesis, University of Massachusetts Amherst, Department of Computer Science, 2009.

[11] N. Russell and W. van der Aalst. Evaluation of the BPEL4People and WS-HumanTask Extensions to WS-BPEL 2.0 using the Workflow Resource Patterns. Technical report, 2007.

[12] T. J. Schriber and D. T. Brunner. Inside simulation software: inside discrete-event simulation software: how it works and why it matters. In *Proceedings of the 33nd conference on Winter simulation*, WSC '01, pages 158–168, Washington, DC, USA, 2001. IEEE Computer Society.

[13] Wikipedia. Anylogic — wikipedia, 2010.

[14] A. Wise. Little-jil 1.5 language report. Technical Report 2006-051, University of Massachusetts Amherst, October 2006.

[15] A. Wise. Jsim agent behavior specification. http://-laser.cs.umass.edu/documentation/jsim/language.html, 2009.

[16] A. E. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton. Using little-jil to coordinate agents in software engineering. In *Automated Software Engineering*, pages 155–164, 2000.

[17] H. Zhang, B. Kitchenham, and D. Pfahl. Reflections on 10 years of software process simulation modeling: a systematic review. In *Proceedings of the software process (ICSP'08)*, pages 345–356. Springer-Verlag, 2008.