Process Definition Language Support for Rapid Simulation Prototyping

Mohammad S. Raunak and Leon J. Osterweil

Department of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
{raunak, ljo}@cs.umass.edu

Abstract. This paper suggests how an appropriately designed and architected process definition language can be an effective aid to the rapid generation of simulations, which are, in turn, capable of providing important insights. The paper describes how the features of the Little-JIL process definition language helped in the rapid generation of simulations that shed important new light on the effectiveness of various collusion strategies in influencing the outcomes of various auction approaches. The paper describes how Little-JIL's approach to modular reuse and its separation of process concerns both turn out to be of particular value in supporting rapid prototyping. The simulation results obtained are themselves interesting, as the paper also suggests that the auction idiom is highly relevant to resource allocation in software development. Thus, the insights gained into the efficacy of various collusion approaches have particular relevance to software process research.

1 Introduction

There is currently considerable interest in using process definitions as the basis for important decisions about such matters as resource allocation, coordination of agents, and procedural issues. Because of the importance of such matters, technologies that are effective in supporting the precise, clear, and complete definition of processes seem to have broad and important applicability. Further, it seems particularly important to evaluate such process definitions carefully to be sure that they are correct and effective, which can then become the basis for their systematic, iterative improvement. We have long argued that software process development has a close parallel to application software development [14, 15]. This parallelism suggests that tools and techniques used in the latter should be expected to be applicable also to the evaluation of processes. From this analogy we know that there are two major types of approaches to evaluation, namely static and dynamic analysis. In earlier papers we have described our work in the static analysis of processes [7]. This work has shown that static analysis can be an effective approach to identifying defects in processes. This paper addresses a dynamic analysis approach, namely simulation.

Simulation seems to be a particularly useful approach to the evaluation of processes. Process simulations can support the quantitative determination of flow times

for executions of processes, the impacts of making certain resource allocation decisions, and the projected behavior of processes under various hypothesized loading conditions. Various authors have previously appreciated this, and have already begun to investigate technologies that support process simulation, and the effectiveness of these technologies [11]. While we agree with the value of this approach, our own work has increasingly demonstrated that it is often difficult to determine just which simulations best provide desired insights. Indeed, our work suggests that one set of simulation runs often raises as many questions as it answers, and inevitably leads to the desire and need for many subsequent sets of different simulation runs. Thus, because it is often the case that the analyst is unclear about the exact process that she/he wants to study, we have found that it is helpful to be able to rapidly create prototype simulations as a key support for the process of rapid exploration of the questions that arise as simulation runs suggest further sequences of elaborative simulations of processes. In grappling with the problem of how best to support the need for rapid process simulation generation, we have concluded that the appropriate process language architecture and execution tools are very important in this task.

We note that this observation itself mirrors similar observations that have been made by developers of simulations in application software domains. Simulations have been shown to be valuable in providing precise answers and insights when the questions and issues are narrowly drawn. Getting the questions appropriately narrowed down, however, can take considerable effort, often in the form of evaluation of sequences of preliminary simulations. Thus the value of being able to create simulations rapidly has been previously understood [19]. This points to the value of flexible simulation generation aids, such as the one that we will describe in this paper.

While our work seems to support the value of automated process simulation capabilities, it has also shown that simulation generators can lead to simulations that are largely interpreted, and consequently inconveniently slow. Our work has demonstrated that, while it is helpful to have rapid simulation generators to help identify the precise simulation that can provide results of great interest, once the simulation has been identified, it seems important to then code the simulation in a compilable language in order to assure that massive quantities of simulations needed for reliability of results, can be executed in acceptably short amounts of time.

All of the above has reinforced in our minds the desirability of an executable design language for processes. In our work we used our Little-JIL process definition language to specify simulation designs. Our Juliette interpreter was used to run these preliminary simulations, emphasizing the value of an executable design. After suitable preliminary simulations, we then moved on to the need to run massive amounts of simulations, at which time we used the Little-JIL designs as the basis for facilitated coding of the desired simulations in Java.

In this paper we describe our work, not only emphasizing the value of an executable process definition language, but also emphasizing how the particular separation of concerns in our language greatly facilitated the rapid generation of preliminary simulations, thereby expediting the definition of the massive simulations that eventually yielded the process analysis results that we sought.

2 Related Work

Simulation has long been used as a powerful tool for analyzing software performance in wide range of application areas. Simulation of software processes has also received a lot of attention over the last twenty years, during which time a wide variety of simulation approaches have been suggested and explored. Kellner et al. [11] provides a nice overall picture of the work in software process simulation, focusing primarily on the diversity of approaches to process simulation, addressing the different issues, and suitability of various approaches to dealing with them. In particular, [11] talks about eight different simulation approaches and language categories including state based process modeling, discrete event simulation and system dynamics. [6] shows how simulation can be of benefit in supporting software process improvement in the context of such approaches as the Capability Maturity Model (CMM). [18] provides some experience-based insights into the effectiveness of knowledge based simulation (KBS) and discrete event simulation (DES) of processes. [13] presents the need and usefulness of combining both discrete event and system dynamics approaches to simulation while simulating software processes.

Although researchers have not looked at issues related to rapid simulation prototyping for software processes, the need has been established in other system simulation domains. For example, [19] discusses the usefulness of rapid simulation and software prototyping for the architectural design of embedded multiprocessor systems.

There is ample literature studying auctions from the various perspectives of Economics, Management Science, Operations Research and Computer Science. Economists mainly look at the auction mechanism from the game theoretic perspective and try to identify optimum price determination that maximizes the utility of the seller and/or the bidder. They also try to reason about the behaviors of the bidders and their impact on the outcome of the auctions. Although most of these analyses are based on probabilistic models, a small but influential trend has been to study market mechanisms through laboratory experiments using discrete event simulation. The work described here is in that spirit.

Computer Scientists and Management Scientists have looked at different variations of auctioning mechanisms for multiple items [3, 17]. Their primary objective has been to devise auctions that are efficient in the determination of the optimal winner. Other researchers in computer science have looked into the capturing of auction processes with rigorous process language and statically analyzing the auctions to verify correctness and completeness [7].

Identifying vulnerabilities in auctions is a very important issue for better auction designs [12]. Some recent work attempts to identify collusive behavior of bidders by analyzing bidding patterns [1, 2]. These efforts, however, are few in number and limited in scope. Stochastic analyses usually fall short of providing a good picture of the outcome of an auction in a setting where there is the possibility of dynamic behaviors of bidders, and where the opportunity for collusion is present. Researchers have opted for empirical study based upon simulation of auction processes with actual human bidders [8, 9, 10].

3 Our Approach

The vehicle for our exploration of the value of a suitable executable process design language and architecture was our Little-JIL process definition language [21]. One of the key architectural features of Little-JIL is its separation of concerns [5]. In particular, the most noticeable feature of a Little-JIL process definition is its visual depiction of the agent coordination aspect of a process (to be described in more detail shortly). Equally important, however, are the definitions of agents and their behaviors, and artifacts and their flows. Neither of these process concerns is explicitly represented pictorially in Little-JIL. While this makes them less immediately noticeable, they are no less important. Of particular importance for this research is the fact that these concerns are separately defined, and separately modifiable.

To understand the importance of the separation of these concerns to the need for rapid prototype generation, we employed Little-JIL as a vehicle for the study of processes in the domain of auctions. An auction is a form of price negotiation that is usually a highly decentralized activity. Participants in auctions are essentially distributed agents coordinated to achieve a goal. There is a large literature addressing the enormous variety of different kinds of auctions. Indeed there are probably at least thousands of different kinds of auctions [12], all of which have somewhat different characteristics. The different kinds of auctions have been devised in a continuing attempt to find the most efficient ways to arrive at an accurate determination of the fair value of a commodity.

It is important to also note, however, that much of the diversity in types of auctions, and the continued lively investigation of auctions, is aimed at understanding the effects of various kinds of collusions among bidders. While there is a considerable amount known about how various different auction strategies may be vulnerable to, or resistant to, different forms of bidder collusions, much more needs to be known. We believe that simulations of various kinds of collusive activities operating in the context of different kinds of auctions have the potential to yield these important understandings.

Our research entailed the use of Little-JIL as a vehicle for supporting the rapid generation and execution of simulations of the different ways in which different auctions responded to bidder collusions. As will be seen, the Little-JIL architecture proved to be particularly useful in this work, as the pictorial coordination concern proved to be effective in defining the different auction processes clearly, precisely, and completely, while the agent definition concern, independently defined (in this case using Java), proved quite effective in defining different collusion strategies. Our Juliette process interpreter, drawing upon the combined coordination and agent definitions, supported the simulation of the different auctions.

It is worth noting here that the selection of auction processes for this research is far from irrelevant to software process concerns. We believe that the auction idiom, and indeed the auction vehicle itself, especially as elaborated using notions of agent collusion, are quite relevant to software development process concerns. We note, for example, that a significant aspect of the software development process involves the allocation of resources or agents to the various development tasks. The effective determination of the most effective allocation of agents (e.g. designers, programmers,

testers) might well be modeled, and indeed carried out, as an auction, where the bidders are software engineers bidding for specific tasks. Communications among these agents should be expected, as is the case of collusive bidders in an auction process, although communication among software developers is generally useful in arriving at effective task assignment, in contrast to the situation in auctions. Thus, our focus on auctions is more than simply illustrative of our ideas about process language architecture, but also seems relevant to the development of superior software development agent allocation strategies.

3.1 The Little-JIL process language

Little-JIL is a process definition language [5, 21] that, along with its interpreter Juliette [4], supports specification, execution, and analysis of processes involving multiple agents. In this work, we used Little-JIL to capture and simulate auction processes and agent interactions. As noted above, the most immediately noticeable aspect of a Little-JIL process program is the visual depiction of the coordination specification of the process. This component of the Little-JIL process program looks initially somewhat like a task decomposition graph, in which processes are decomposed hierarchically into steps. The steps are connected to each other with edges that represent both control flow and artifact flow. Each step contains a specification of the type of agent needed in order to perform the task associated with that step. Thus, for example, in the context of an auction, the agents would be entities such as the auctioneer, the bidders, as well as, potentially, the manager setting up collusion amongst the bidders. The collection of steps assigned to an agent defines the interface that the agent must satisfy to participate in the process. It is important to note that the coordination specification includes a description of the external view and observable behavior of such agents. But a specification of how the agents themselves perform their tasks (their internal behaviors) is NOT a part of the coordination specification. The behaviors of agents are defined in a separate specification component of the Little-JIL language. More will be said about this shortly. But it is important to note that Little-JIL enforces this sharp separation of concerns, separating the internal specification of how agents carry out their work, from the specification of how they coordinate with each other in the context of carrying out the overall process. In particular, the definition of a particular specific auction is defined separately from the definition of how the bidders might collude with each other.

The central construct of a Little-JIL process is a step. Steps are organized into a hierarchical tree-like structure. The leaves of the tree represent the smallest specified units of work, each of which is assigned to an agent that has characteristics consistent with those defined as part of the definition of the step. The tree structure defines how the work of these agents will be coordinated. In particular, the agent assigned responsibility for executing a parent node is responsible for coordinating the activities of the agents assigned to execute all of the parent's children.

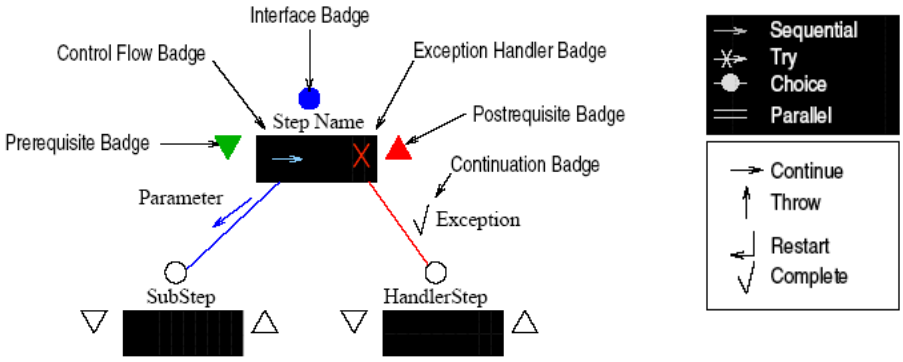

Fig. 1a. A Little-JIL step construct **Fig. 1b.** Control flow and continuation badges

Figure 1a shows the graphical representation of a Little-JIL step with its different badges and possible connections to other steps. The interface badge is a circle on the top of the step name that connects a step to its parent. The interface badge represents the specification of any and all artifacts that are either required for, or generated by, the step's execution. Of greater importance for the work described in this paper, the interface badge also represents the specification of any and all resources needed in order to support the execution of the step. Chief among these resources is the single resource designated as the step's execution agent. Below the circle is the step name. A step may also include pre-requisite and/or post-requisite badges, which are representations of steps that need to be executed before and/or after (respectively) this step for the proper performance of the step's execution. Inside the central black box of the step structure, there are three more badges. On the left is the control flow badge, which specifies the order in which the child substeps of this step are to be executed. A child of a step is connected to the parent by an edge emanating from the parent and terminating at the child. Artifact flows between the parent and child are indicated by annotations on this edge.

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 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 is itself 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. This edge also bears an annotation indicating the type of exception handled.

In the middle of the step bar goes a "lightning sign" (not shown in Fig. 1), which represents the message handling capabilities of the step. Attached to this badge by message handling edges are any and all handlers defined to deal with messages that may emanate from any step in the process definition. 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 un-

scoped capability). If there are no child steps, message handlers, or exception handlers, the corresponding badges are not depicted in the step bar.

One of the important features of the language is its ability to define control flow. There are four different non-leaf *step kinds*, namely "sequential", "parallel", "try" and "choice". Children of a "sequential" step 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 "try" step attempts to execute its children one by one starting from the leftmost one and considers itself completed as soon as one of the children successfully completes. Finally a "choice" step allows only one of its children to execute, with the choice of which child being made by the agent assigned to execute the step.

The *pre-requisites* and *post requisites* associated with each step act essentially as guards, defining conditions that need to hold true for a step to begin execution or to complete successfully. *Exceptions* and *handlers* are control flow constructs that augment the step kinds. The exceptions and exception handlers work in a manner that is similar in principle to the way in which they work in well known contemporary application programming languages. Exceptions indicate an exceptional condition or error in the process execution flow, and handlers are used to recover from, or fix, the consequences of those situations. When an exception is thrown by a step, it is passed up the tree hierarchy until a matching handler is found. There is control flow semantics involved with handler steps to indicate how the program flow will continue once a raised exception has been handled by the defined handler. Figure 1b shows four different types of continuation semantics for handlers. With these semantics, a process definer can specify whether a step will continue execution, successfully complete, restart execution at the beginning, or rethrow the exception for a higher level parent step to handle.

As noted above, a complete Little-JIL process definition also contains definitions of *artifacts* and *resources* to complement this coordination definition. *Artifacts* are entities such as data items, files, or access mechanisms that are passed between parent and child steps. They provide information required for execution of a step and can be used to carry results of the step execution back to the parent. Again, as noted above, the artifact definition, indeed the specification of the type model used to support artifact definition, is a separate concern in Little-JIL, and is orthogonal to the coordination definition.

In an analogous way, the resource (and thus agent) definition is also separate from, and orthogonal to, the Little-JIL coordination definition. Specifically, how an agent carries out a particular task is independent of the coordination dictated by the process. Of course, however, the outcome of a process is influenced by the behaviors of the agents, which are, in turn, specified within the resource model. Consequently, the outcome of a process is similarly affected by the way in which specific resources are bound as agents to the various individual process steps at various points during process execution.

3.2 Auction Processes

We opted to investigate different auction processes and combinations of different bidder behaviors in a potentially collusive environment. In the course of our investigations of how these behaviors affected auction outcomes, we wound up modeling many different types of auctions, including open-cry or ascending bid (English) auctions, double auctions, first-price sealed-bid auctions and finally repeated sealed-bid auctions. Space does not permit us to describe the all the details of these different types of auctions. The interested reader can find these details in [16]. But it is important to sketch out some of their salient properties as this helps us to explain how certain properties of Little-JIL were particularly useful in supporting the rapid creation of prototypes of auctions of these types.

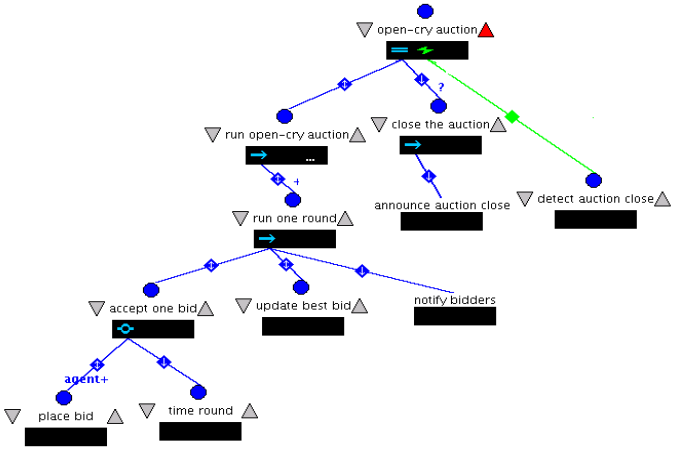

Fig. 2. An Open-cry (Ascending bid/English) auction

In an ascending-bid (English) auction, the auction type that is most commonly depicted in movies and novels, the price is successively raised by the auctioneer until only one bidder remains, and that bidder wins the object paying the final price that was bid. A Little-JIL coordination definition of this type of auction is depicted in Figure 2.

A double auction differs from an English auction in that both buyers and sellers submit bids in parallel. The auctioneer opens this bidding, and then periodically closes the bidding, completing a "round". At the end of a round, the auctioneer identifies matches between the bids of buyers and sellers, finalizing the sale of items so matched. Figure 3 shows a Little-JIL coordination definition that describes such a double auction process. As you can see "double auction" is a parallel activity between running the auction and checking to make sure that the auctioneer has not called a stop to the auction. The step "run double auction" has a pre-requisite step "check auction close" that identifies whether the next round should be placed or not.

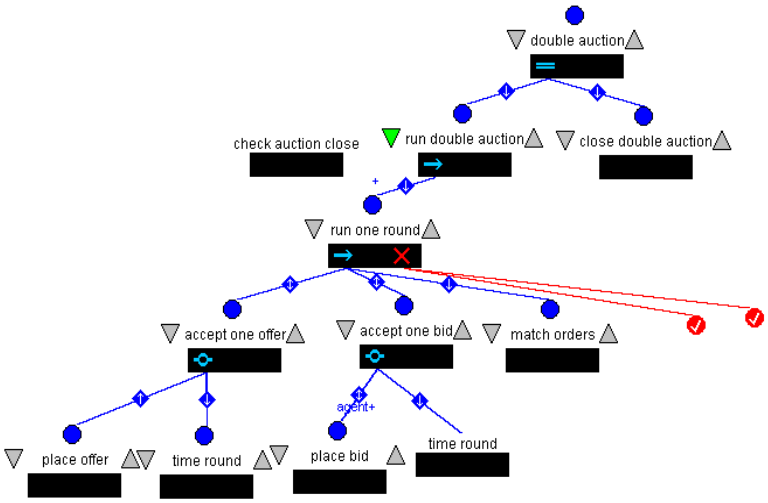

Fig. 3. A double auction process

The important point to note here is that there are some common activities in these auction processes. For example, the auctioneer closes the auction after a predefined time period. Placing of bids is common to all the auctions. The auctioneer needs to decide on a winner by processing the bids. Because of this, one should expect that there ought to be process modules from one auction definition that are reusable, and reused in other auction process definitions. Little-JIL encourages and supports such reuse, as can be seen by examining these process coordination definitions. For example, the “accept one bid” module of the double auction process is taken “as-is” from our ascending-bid/open-cry auction process.

Examples of more extensive successful reuse can be seen in additional auction process definitions. Our work continued with the definition of sealed bid auctions. In a first-price sealed-bid auction, each bidder independently submits a single bid without knowing others' bids, and the object is sold to the bidder with the highest bid. The bidder pays his price (first price) to get the object. Auction of this type are currently very common and popular.

A repeated sealed bid auction is an important variant of this type of auction. It is a series of sealed bid auctions where auctioneer announces the results of the auction after every round, and then initiates a new auction for new batch of goods or services that are essentially identical to those just sold. Governments and large corporations carry out much of their procurement activities through exactly this kind of repeated sealed-bid auctions. Because of the enormous economic importance of such auctions, they have been the subject of much analysis, much of which has focused on their resistance, or vulnerability, to certain kinds of bidder collusion. The bidders, after receiving the announcement of the outcome of a round, can potentially attempt to collude or decide on their bids individually. In either case, the bidders place their bids

and auctioneer collects the bids, processes them to identify the winner and announces the winner before initializing the new round.

For space constraint, we are not showing the Little-JIL coordination definition of a repeated sealed bid auction. However, it is important to note that this process reuses the entire definition of a single-round sealed bid auction, in a striking demonstration of reuse. The single round sealed bid auction, in turn, reuses some steps from the open cry auction definitions.

There are other features of Little-JIL that foster reuse and rapid prototyping, but they require familiarity with other features of the language that we address now. Specifically, it is important to note that some edges are annotated with cardinality symbols, for example the *agent+* notation on the edge from the "accept one bid" step to the "place bid" step. This notation represents the r*esource bounded cardinality* feature of Little-JIL process descriptions. The child step of an edge containing such a notation is instantiated once for each of the agents available as an active bidder at the time of instantiation of that step. Thus, for example, if we specify that the agent for a bidding step must be collusive, then one step will be created for every bidder whose agent behavior (as defined in the resource factor of the Little-JIL process definition) is defined to be collusive. A bidding step will be instantiated only for the collusive bidders and each of them will be given the task of bidding in that step. In contrast, if a step's agent specification specifies only bidders, then that step will be instantiated for all bidders, both collusive and non collusive. If an edge is annotated with '*?*', then that step will execute only if an agent satisfying the requested characteristics is available. Thus, the step may or may not execute at all. For example, in the repeated sealed-bid auction process there is a collusion step connected to its parent by an edge annotated with a "?" indicating that it will get executed only if there exists collusive bidders in that auction round.

One net effect of these properties of Little-JIL is that a given fixed auction process can be executed, and evaluated, against a variety of different bidder collusion scenarios very straightforwardly. Indeed, the coordination definition may not need to be changed at all in order to evaluate the resistance of a particular auction to a variety of collusive threats.

Conversely, it is correspondingly straightforward to evaluate the relative resistances of various auctions to a fixed collusive threat. Because the code for defining how agents will collude with each other is contained largely in the agent definition factor of Little-JIL, it is also highly reusable across the range of different auction processes. In our work, we have been able to plug in the agent code written to support one auction process, and use it to support a different one, with minimal or no changes to the agent code. Thus, Little-JIL's separation of concerns allows us to quickly change an overall auction process, by changing either the auction or the agent behavior, but then reusing the other part with little or no change. This has enabled us to create and evaluate a wide range of auction processes very rapidly.

3.3 Modeling Collusions

We have already noted that agent behaviors, such as collusions in an auction, are defined in Little-JIL as part of the agent definition factor of the language. Thus, our repeated sealed bid auction did not explicitly model the details of the collusive behavior. It is expected that these behaviors may best be defined in other languages, specifically those that are more strongly algorithmic or computational. On the other hand, there is no reason why agent behaviors cannot be defined as Little-JIL coordination, using the power and convenience of the Little-JIL coordination language itself. Thus, as an example, Figure 4 shows one possible definition of collusive behavior, which we have used in many of our simulations.

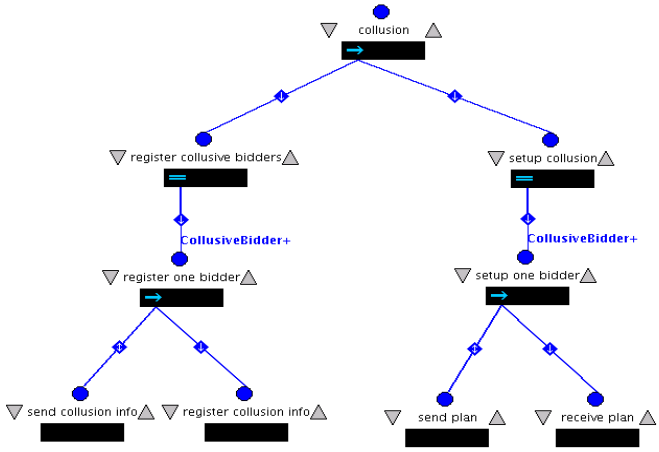

Fig. 4. A collusion process example

The collusion protocol presented here describes the scenario where each collusive bidder registers with a collusion manager. While registering, the bidders submit their valuation for the auctioned object to the collusion manager. The manager then sets up the collusion by sending a collusion plan back to the registered bidders. The plan includes what each of the bidders should bid in that round. It is important to note that we indeed used Little-JIL to define a number of collusions in our preliminary auction research, as the language's encouragement and support of reuse facilitated the rapid creation of prototype auctions that differed from each other only in modest perturbations of collusive behavior.

Thus, for example, the strategy used by the collusion manager to decide on the instructed bid can, in the case of the collusion shown in Figure 4, be made independent of both the auction process and the overall collusion strategy itself. This allows us to quickly create a new prototype for a different type of collusion while running the same auction and using bidders who intend to collude in the same way.

4 Experiences and Challenges

4.1 Auction Results

After a few iterations aimed at identifying the simulation that seemed to be most promising for investigation, we zoomed into an intensive and detailed examination of the repeated sealed bid auction where bidders have the opportunity to collude. We ran several experiments using our process simulation vehicle. Our aim in the experiments was to see how the number of collusive bidders impacts the outcome of an auction. We were also interested in investigating the dynamics of multiple colluding rings operating in a repeated auction environment. We developed automated agents with bidding strategies based on game theoretic models. We allowed for changes of behavior amongst the bidders. A non-collusive bidder can become collusive and join a ring after a few rounds of auction based on the history information made available to the bidder. We modeled our bidders as risk neutral agents placing their bids according to a pareto-optimal bid producing a Nash equilibrium [20]. The bidders were made increasingly complex in later experiments. The bidding strategy was primarily modeled as a decision problem influenced by a lot of factors and parameters. If there are multiple colluding rings present, we allowed colluding bidders to switch rings under certain conditions. As noted earlier, we modeled the collusion with communication amongst the bidders through a center, the collusion manager. The center's role was to decide on the profit sharing mechanism of the colluding bidders and instruct participating bidders to bid according to the prescribed strategy. Here we present a very brief, but representative, summary of our experimental results to demonstrate the usefulness of such a simulation study. A more detailed discussion about the modeling of the bidding strategy and the results of different experiments we performed is to be presented in subsequent papers.

In this experiment, we used a fixed number of ten non-collusive bidders participating in the auction. However, after a fixed number of rounds, we updated bidder behavior and made a non-collusive bidder collusive. As the auction progressed through subsequent rounds, more and more bidders became collusive. Our object was to identify whether the collusive ring needs to achieve some kind of threshold size in a sealed-bid auction in order to consistently be successful in rigging the auction. We observed that with a specific auction model and bidder behaviors, the collusion starts to take over when around 70% of the bidders participate in the collusion. In another experiment, we have been investigating the conditions under which one dominant colluding ring drives other rings out of competition. The initial results have shown some interesting trends. We will present the details of our experiment setup and evaluations in subsequent papers.

4.2 Simulation Experience

The process definition and execution framework supported by Little-JIL and Juliette facilitated our efforts, and enabled us to execute hundreds of cycles of process definition, execution, evaluation, and evolution in a short space of time. The simulations we developed often provided intriguing results, causing us to feel the need for validation. Thus, one of our early activities entailed the generation of some large scale simulation outputs that could be verified against analytic results. The validation of these early simulation results encouraged us to go on to simulations that entailed complex, yet realistic, collusions that are not amenable to analytic verification. These simulations seem to add to the body of knowledge about the effectiveness of various collusions against particular auctions.

In order to obtain these results, we felt it was necessary to perform massive amounts of simulation runs of various configurations of colluding bidders, against the auction processes that we had decided to study. It was not initially obvious, however, which type of auction, and what type of collusion, was worth this evaluation through massive quantities of simulation runs. We felt that there was a need for the flexibility of rapid changes of process, collusion and bidder strategies necessary in order to identify the specific auctions, collusions, and bidder behaviors that were likely to be of most interest and value in auction research. The factoring of process coordination, agent assignments and actual agent behavior that we leveraged out of the Little-JIL/Juliette framework supported the separation of concerns in a process language framework that enabled the considerable amount of exploration of this sort that we found to be needed.

Once the specific simulations that needed to be evaluated through massive experimentation became clear as the result of a considerable period of this preliminary evaluation, the drawbacks of an interpreted language became apparent. The auctions that we wanted to simulate extensively entailed ten or more bidders, with the process itself consisting of more than eighty steps for every auction round. Dozens of auction rounds were necessary.

As Little-JIL is an interpreted language, executing processes with Juliette is naturally far slower than would be the execution of the same processes programmed in a compiled language. Moreover, Juliette was designed to support distributed process execution. This distributive support is accomplished through a lot of remote method invocations (rmi) which incurs large network communication time, which in turn makes the executed process slow. With real humans in the loop for placing the bids from distributed terminals, the simulator created with this infrastructure can be sufficient to produce realistic results. However, if one wants to focus on producing massive simulation results in a short period of time through automated agents, automated process execution falls short of providing that level of efficiency. In our experiments with ten bidders, each round of sealed-bid auction took about two to three minutes to finish depending on the number of bidders colluding. More colluding bidders result in an increased number of total steps for that auction round. In an experiment where the number of collusive bidders increases with time, the execution time for each auction also rose sharply. Toward the end of a thirty round experiment, each auction cycle took up to six minutes on average to finish. However, we did not intend to produce

massive simulation with the distributed process execution framework of Little-JIL/Juliette. At this point, we used our process definition as the architecture of the intended simulation engine and rapidly coded out the simulator in a compiled language, Java. The beauty of this switching was the ability to reuse a lot of agent code. As agents were a separate concern in our process language and were written in Java in the Little-JIL/Juliette framework, it was easy to plug in the agent code in later simulations.

4.3 Process language experience

We have gathered some important insights regarding process programming in general and Little-JIL/Juliette framework, in particular, while implementing this simulation infrastructure. We have used Little-JIL as an executable design language to create a meta-model infrastructure to build simulators.

Little-JIL provided us with rich process notations to depict the architecture of the simulation hiding unnecessary details. We were able to capture the behaviors of a range of auction processes quite accurately, with relatively little effort being put into creation of the different processes. For example, Little-JIL's concurrency control features were particularly useful in defining the simultaneous placing and accepting of bids by the bidders and auctioneer respectively. Little-JIL also helps to describe certain steps succinctly. The agent bounded cardinality in *place bid* step succinctly, yet effectively, selects for instantiation steps executed only by agents available to carry out the work. Specifying different attributes of the agents allowed us to assure the selection of the agent appropriate to perform any particular task at any particular process execution instance.

We note (in passing, only to save space) that Little-JIL also incorporates some timing semantics. In our repeated sealed-bid auction process, we have used a deadline construct (a clock-face on a step interface) on the *place-bid* step. This indicates that the step has to be completed by the agent within a specified period of time. Otherwise it will be retracted and a *deadline expired* exception will be thrown. The inclusion of deadline semantics in the language supports the definition and evaluation of an even richer collection of auction processes.

Juliette, the Little-JIL interpreter, takes a little-JIL process and executes it in a way that is assured to be completely consistent with the Little-JIL semantics, through the use of finite state machine semantics that are used to define Little-JIL and also drive Juliette. Juliette, moreover, deals with a resource manager that is the repository of all resources (and, therefore agents) that are available for participation in the execution of the process. Thus, Juliette has the ability to acquire resources required to complete each step (agents etc) incrementally, and in real time, as the process execution proceeds. Juliette also manages the numerous data that flows from step to step throughout the process.

One shortcoming of the Little-JIL coordination language is the absence of semantics to support specification of artifact flow between sibling steps. As noted above, artifact flow in Little-JIL is defined to take place between parent and child. But this posed a continuous problem in our auction processes, as it complicated the represen-

tation of how bids flow from bidder to auctioneer, and how results flow from auctioneer to bidder. We have used the message passing semantics of using reactions and reaction handlers in Little-JIL to represent this type of lateral dataflow in our simulated processes. New versions of Little-JIL are due to incorporate features that would alleviate this problem, and these features are clearly necessary to facilitate research of the type that we have been describing.

5 Concluding Remarks

Little-JIL, a rigorous process language, has been used to define a wide range of processes. Static analyzers have been applied successfully to reason about processes defined by this language. In this paper we have utilized the factored nature of the Little-JIL language, its separation of concerns, and its flexible execution environment, to develop a simulation framework to perform dynamic analysis of a specific type of distributed process, auctions and collusions in auctions. We demonstrated the utility of an executable process language in providing support for rapid simulation prototyping. We have also presented our case for the need of quick simulation development for identifying the right focus that needs further investigation. Our findings pointed out the issues important for process based simulation development. We investigated a number of simulations and finally zoomed into reasoning about auction outcomes in a repeated sealed-bid auction scenario with the presence of collusive bidders.

Although auctions have been extensively studied analytically in Economics and Operation Research areas, there are certain characteristics of auctions that are difficult to analyze analytically. Researchers have used laboratory experiments with small numbers of human bidders to study some such characteristics [8, 9, 10]. In our study, we utilized the theoretical findings of auction researchers to model automated bidders to run large numbers of auction simulations with varied parameters. We have found that in a repeated sealed-bid auction with risk neutral bidders, there seems to be a certain threshold governing a collusion's effectiveness in the auction. We also identified the effect of collusion when multiple rings of differing size are present in an auction. These insights should help us better understand the effectiveness of collusion in auctions and in turn, allow us to design better auction processes that are resistant to collusive behavior.

The demonstration of auction analysis through simulation indicates the clear potential for dynamic analysis of software processes. Auction processes are economic activities used widely for efficient resource allocation. This type of resource allocation and task assignment are, however, also common activities in software development processes. Like an auction environment with collusion, side communication amongst agents in a software process and bidding for work is not uncommon at all. However, the exact software process to study may require iterative prototyping of the process simulation. It is therefore our contention that there is a lot of merit in using flexible process language to produce executable simulation designs through rapid prototyping.

Acknowledgements

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032, by the U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231, and by the National Science Foundation under Grant No. CCR-0204321. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, the U.S. Dept. of Defense, the U.S. Army, The National Science Foundation, or the U.S. Government.

Prof. Abhijit Deshmukh helped us a lot in guiding us towards the more interesting auction processes to study. His valuable input allowed us to model the collusions and bidding strategies of the automated bidders. Conversations with Prof. George Avrunin were also very helpful in this regard. Sandy Wise and Aaron Cass have been very helpful in providing insights and advice about the Little-JIL language, and the Juliette interpreter. Ethan Katz-Bassett and Amr Elssamadisy helped us during the simulation modeling and implementations of the initial auction processes.

References

1. Aoyagi, M.: Bid rotation and collusion in repeated auctions. Journal of Economic Theory Vol. 112 (1) (2002) 79-105
2. Bajari, P., Summers G.: Detecting collusion in procurement auctions. Antitrust Law Journal Vol. 70 (2002) 143-170
3. Byde, A.: A comparison among bidding algorithms for multiple auctions. Technical Report, Trusted E-Services Laboratory, HP Laboratories Bristol (2001)
4. Cass, A.G., Lerner, B.S., McCall, E.K., Osterweil, L.J., Sutton Jr., S.M., Wise, A.: Logically central, physically distributed control in a process runtime environment. Technical Report No. UM-CS-1999-065, University of Massachusetts, Department of Computer Science, Amherst, MA (1999)
5. Cass, A.G., Lerner, B.S., McCall, E.K., Osterweil, L.J., Sutton Jr., S.M., Wise, A.: Little-JIL/Juliette: A process definition language and interpreter. In: Proceedings of the 22nd International Conference on Software Engineering, Limerick, Ireland (2000) 754-757
6. Christie, A.M.: Simulation in support of CMM-based process improvement. Journal of Systems and Software, Vol. 46(2). (1999)
7. Cobleigh, J.M., Clarke, L.A., Osterweil, L.J.: Verifying properties of process definitions. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA2000), Portland, OR (2000) 96-101
8. Isaac, R.M., Plott, C.R.: The opportunity for conspiracy in restraint of trade: An experimental study. Journal of Economic Behavior and Organization Vol. 2 (1981)
9. Isaac, R.M, Valerie, R., Arlington W.W.: The effects of market organization on conspiracies in restraint of trade. Journal of Economic Behavior and Organization, Vol. 5. (1984) 191-222

10. Isaac, R.M., Walker, J.M.: Information and conspiracy in sealed bid auction. Journal of Economic Behavior and Organization, Vol. 6. (1985) 139-159
11. Kellner, M.I., Madachy, R.J., Raffo, D.M.: Software process modeling and simulation: Why, what, how. Journal of Systems and Software, Vol. 46(2) (1999)
12. Klemperer, P.: Auction theory: A guide to the literature. Journal of Economic Surveys, Vol. 13(3) (1999) 227-286
13. Lakey, P.B.: A hybrid software process simulation model for project management. In: Proceedings of the Software Process Simulation Modeling Workshop, Portland, OR (2003)
14. Osterweil, L.J.: Software processes are software too. In: Proceedings of the Ninth International Conference of Software Engineering, Monterey, CA (1987) 2-13
15. Osterweil, L.J.: Improving the quality of software quality determination processes. In: R. Boisvert, (ed.): The Quality of Numerical Software: Assessment and Enhancement. Chapman & Hall, London (1997)
16. Milgrom, P.R., Weber, R.J.: A theory of auctions and competitive bidding. Econometrica, Vol. 50(5). (1982) 1089-1122
17. Sandholm, T.: Algorithm for optimal winner determination in combinatorial auctions. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), Stockholm, Sweden (1999) 542-547
18. Scacchi, W.: Experience with software process simulation and modeling. Journal of Systems and Software (1999)
19. Thuente, D.J.: Rapid simulation and software prototyping for the architectural design of embedded multiprocessor systems. In: Proceedings of the 19th annual conference on Computer Science, San Antonio, Texas (1999) 113–121
20. Vickrey, W.: Counterspeculation, auctions, and competitive sealed tenders. Journal of Finance, Vol. 16 (1961) 8-37
21. Wise A.: Little-JIL 1.0 language report. Technical Report No. UM-CS-1998-024, Department of Computer Science, University of Massachusetts, Amherst, MA (1998)