A Reference Model for Simulating Agile Processes

Ian J. De Silva
desilva@cs.umn.edu
Sanjai Rayadurgam
rsanjai@cs.umn.edu
Mats P. E. Heimdahl
heimdahl@cs.umn.edu
Department of Computer Science and Engineering
University of Minnesota

ABSTRACT

Today’s software development projects must respond to a constantly changing marketplace, fierce competition, and rapid technological innovation. Agile development processes are popular when attempting to respond to these changes in a controlled manner; however, selecting an ill-suited process may increase project costs and risk. Before adopting a seemingly promising agile approach, we desire to evaluate the approach’s applicability in the context of the specific product, organization, and staff.

Agile processes require high levels of interpersonal communication and trust; thus, individual behaviors of team members may significantly impact project outcomes. To the best of our knowledge, no current process modeling and evaluation technique integrates the product, project, and process with individual behavior in a single model without tightly coupling the process and product. In order to construct an integrated process model for process evaluation, we require a vocabulary for expressing the model’s essential constructs and relationships.

To address this gap, we introduce a process simulation reference model that provides the constructs and relationships for capturing the interactions among the individuals, product, process, and project in a holistic fashion—a necessary first step towards a process evaluation environment for agile processes.

1. INTRODUCTION

Agile development processes are intended to allow companies to respond to market change in a controlled manner [2]. However, we do not yet fully understand which agile techniques or activities are successful and why they are successful; little data and few models are available to support investigating and evaluating these techniques. This lack of information makes process selection particularly difficult. Rather than blindly adopting some favored or fashionable Agile process, which could lead to goal violation (e.g., unexpected budget overruns, schedule delays, or missing functionality), it would be highly desirable to be able to evaluate candidate processes before adoption, considering the specific project context—i.e. the people, product, and project—in the evaluation. Our long-term goal is to reduce the risks associated with process adoption through a priori process evaluation within an environment that accounts for project context. Simulation provides a sufficient basis for such an environment as it can capture the dynamic behaviors of the system.

There are a number of existing process simulation environments, however, these environments generally fail to separate the product from the process within their simulation model and many treat the individual as a simple information processor—ignoring individual and non-process-compliant behavior. Because we wish to evaluate processes, we need to extricate the process under evaluation from product concerns. Moreover, agile techniques particularly emphasize the importance of people in determining the success of a software development effort [5, 7, 10]. We, therefore, desire to explicitly characterize individual behavior beyond simple data processing and message generation. We hypothesize that if people, and by extension, individual behaviors are truly important to project outcomes, then modeling and simulating individual behavior in conjunction with the process, product, and project will improve the accuracy of predicted outcomes. As agents provide a natural and intuitive analogy for individuals [9, 14], we focus on agent-based simulation.

As a prerequisite for simulation—both for developing the simulation engine and for modeling a process—we require a reference model that describes the base constructs and relationships that we wish to model. In this paper, we present an agent-based reference model describing the interactions among the people, product, process, and project, thereby laying the groundwork for a simulation-based process evaluation environment.

2. REFERENCE MODEL REQUIREMENTS

In order to develop a process evaluation environment we require a reference model that captures the essential simulation model constructs and their relationships. The reference model can then be implemented by a simulation engine and transformed into a grammar for simulatable input models. Figure 1 illustrates how the reference model fits into our plan to create a process evaluation environment. In this section, we lay out our criteria for the reference model—the ability to model agility and the independence of the process specification—as well as our desired simulation outputs.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 2015 ACM X-XXXXX-XX/XX/XX ...$15.00.
Both the product and process within our reference simulation—

2.2 Process Specification Independence

The project deliverables in some way. This necessitates modeling 

reference model.

customer drives requirements change as well as require-

behavioral component must sup-

lows us to model the behaviors exhibited by each individ-

as well as the team interactions, we require a rich behav-

process priorities. Both aspects must be captured in the 

processes—as lightweight, iterative and incremental soft-

These processes emphasize (1) individuals and their inter-

actions; (2) early and frequent customer feedback elicita-

control over the team resulting in less documentation and higher levels of inform-

communication [17]—with a focus on delivering working software [1, 2]. The objective of this work is to define 

a reference model suitable for expressing simulatable agile process models. Such a reference model must capture these 

aspects of agile processes.

Any experienced software developer will tell you that in-

process requirements change—be it arrivals, removals, or small changes—generally causes rework. Thus, to model 

requirements changes, we must model more than just the 

change. We must model the potential effects of rework on the project deliverables and duration.

Trusting the team empowers the individuals on the team to perform their job however they see fit. To capture this as well as the team interactions, we require a rich behavioral component within the simulation model; one that allows us to model the behaviors exhibited by each individual composing the team even if the behaviors deviate from the process. Indeed, this behavioral component must support heterogeneous entities to allow us to model external stakeholders embedded in the team (e.g. product owners in scrum). Further, because agile processes emphasize low process ceremony, many agile processes prescribe fewer process steps and/or do not define a total ordering of these steps.

Eliciting early and frequent customer feedback is primarily a process concern. However, the feedback gathered from the customer drives requirements change as well as require-

ment prioritization. Both aspects must be captured in the reference model.

In order to elicit feedback faster and reduce process ceremony, agile processes emphasize “working software over compre-

hensive documentation” [1]. This necessitates modeling the project deliverables in some way.

2.2 Process Specification Independence

At this point, we have established our desire to represent both the product and process within our reference simula-

1. Quality sometimes replaces scope in these definitions as seen in [6].
ous credibility [25,27]. Rather than predicting quality using such indirect metrics, we wish to predict the product quality directly. To this end, we require that our simulation output include a product quality metric. While there are a number of metrics for product quality, defect density—the ratio of the number of defects to the number of opportunities to introduce errors—is the metric we choose due to its widespread use [16]. In measuring the defect density we define the number of defects as the number of unfixed defects in the product deliverables at the end of the simulation. The number of opportunities for errors is, at its essence, the scope of the product, be it source lines of code [28] or function points [16]. For consistency, we reuse our previously defined scope metric here.

The metrics defined thus far align with the definitions used in literature [9,21,28], although several authors define additional metrics including the number of generated errors [21] and process productivity [28]. The number of generated errors can be derived from existing metrics. Process productivity is the ratio of the number of units of output (our scope metric) to units of effort [16,28]. We already plan to collect the constituent metrics, thus we could trivially provide this metric if desired.

Thus, our set of outputs are the project duration, effort, disaggregated effort, product functionality, and defect density.

3. EXISTING APPROACHES

There are numerous approaches to a priori process evaluation. One such example is Martin and Raffo’s hybrid-simulation to evaluate the impact of process changes on project duration, effort, and quality [21]. Their model captures a process as a procedural workflow with development artifacts moving through it. In this model, the process activities and product work packages are separated; work packages (one type of artifact in this model) flow into activities and are transformed into new artifacts. Another example is the system dynamics process model used to provide support in balancing new function development and support activities [19]. This approach models development actions as flows between levels (or stocks of completed tasks) with an information network that computes simulation variables based on the values at certain points in the flow. Both discrete event simulation as well as Martin and Raffo’s hybrid approach require full prescription of the process sequence, making it difficult to model situations where there are multiple possible activity orderings to achieve the same outcome. Systems dynamics approaches abstract out the individual completely, preventing analyses that depend on individual behavior [22].

To compare the impacts of team composition and process adoption on the outcomes of an engineering design project, the integrated product team (IPT) work defines both a multi-agent simulation model and simulator that incorporate the individual, team, product, and process into a single model [9,33]. The objective of this model is to support comparison through sensitivity analysis, not to predict project outcomes. This model defines three types of agents: those that complete tasks, those that assign tasks, and those that answer questions. Tasks, which may be process activities or work packages, are arranged into a dependency network, where a task cannot be executed until its predecessors are complete. As we discussed earlier, this merging of process and product concerns is undesirable. Further, in the IPT model, task quality is estimated, but rework is not modeled. This failure to explicitly capture a key reality of projects poses a risk to the accuracy of their predictions, as finding and fixing defects is a key component of the development process. Further, short of rolling in rework as part of requirements changes, this poses additional difficulties—beyond those caused by tight product-process integration when modeling projects where requirements change.

The virtual design teams (VDTs) simulator [15] attempts to capture the emergent behavior and communication of design teams using a hybrid discrete event simulator and multi-agent system. The VDT model treats human actors (agents) as information processors that, as a result of the activity they are performing, generate outgoing messages/process exceptions. Based on its individual properties, an actor selects a message (including task assignments) to process. Tasks are richer in this model, including not only task dependencies but properties to aid in the model’s computations. One such computation triggers rework of the task by the actor. However, rework is isolated to only the task currently being worked on by the actor rather than some previous artifact. In fact, VDT only measures the process quality, not the product quality [15], providing only indirect quality estimates. There have been many versions of the VDT model, and, like the IPT model, these models do not separate the product from the process [20].

Another agent-based simulator, TEAm Knowledge-based Structuring (TEAKS), attempts to support team configuration by predicting team performance [23]. In this simulation, behavior selection is based on fuzzy logic over behavior-influencing characteristics such as personality trends, emotional state, and social traits [23] as well as the assigned task’s difficulty and type [24]. This model focuses on the agent interactions rather than accurate estimations [24], and it is unclear what they do to model the agent’s environment. Like VDT and IPT, there is no separation of process- product- tasks. Further, the tasks are fixed on input, inhibiting the explicit modeling of requirements arrivals.

The Articulator project is an effort to create a meta-model and agent-based simulation to evaluate processes and predict project outcomes [26]. The Articulator’s meta-model—composed of three primary components: (non-human) resources, agents, and tasks—integrates people, process, product, and project concerns within a single model. The components of this meta-model decompose into a number of elements. At the lowest level, a task is a fully-specified action that may describe organizational work or a process activity [26]. Thus, it too fails to separate out the process from the product. The Articulator specifies goal-based agents who select actions by reasoning over predicates at three levels—the domain, task, and strategy spaces—and generating new facts. This limits the model as it cannot express agent preferences toward particular actions [31].

4. A MOTIVATING SCENARIO

Having, thus far discussed our requirements for a reference model and relevant related work, we present a brief motivating scenario, based on one of the author’s experience, of a process and target context that we wish to model for simulation.

A team is going to start a mobile app development project and desire to evaluate a set of processes to determine which is going to best fit their needs. The project is time con-
straining. The product’s requirements decomposition is partially depicted in Figure 2. In this illustration, larger boxes represent work groupings—stories in most agile processes—and smaller boxes represent work packages—small pieces of function required to complete the work specified in the grouping. In our discussion, work packages do not include process activities. Assume that the team is required to produce a suite of automated regression tests to support long-term maintenance.

5. REFERENCE MODEL

In our discussion of the reference model’s requirements, we described a number of constructs that we wish to include as well as some high-level relationships among them. Because we wish to model the behaviors of heterogeneous individuals, we introduce autonomous agents to our reference model. We choose this representation because both agents and humans are (1) bounded problem-solving entities, (2) situated in an environment with limited observability of the environment, (3) autonomous, and (4) reactive [9, 14]. Further, agents provide a way to encapsulate behavior and decision-making, isolating changes and reducing coupling. This does not prevent us from representing entire teams as agents, but in this work we focus on modeling only individuals.

As a consequence of modeling individuals, we must also model interactions. We model such interactions as messages passed from an originating agent to one or more other agents. The VDT model, in addition to communication constructs, defines the concept of a communication channel. The channel introduces a number of properties for communications that capture if the communication will interrupt the recipient and enable reasoning about message priority [15]. This mechanism can also introduce communication delays and provide a way to model globally-distributed teams, therefore we include a communication channel construct in our model.

Because we use agents to represent individuals, we require some concept of an action that an agent can perform, providing a mechanism for the agent to do any number of things, such as generate messages. The agent, then, must be able to choose an action and execute it, exhibiting a behavior. Additional details about the workings of the agent are beyond the scope of this work as our focus is to describe the essential modeling constructs and relationships for process models.

As discussed, we want to decouple product function from process activities. To aid in this endeavor we define a number of constructs. The first, artifacts, represent the deliverables of the project. Work packages specify the smallest unit of useful function that may be added to one or more artifacts. Work packages may be dependent on other work packages, forming a partial ordering. Process activities describe how to perform work specified by a work package, essentially enabling an agent to transform a work package into a feature within a given artifact. They too are partially ordered. Agents, while executing an activity, may generate defects and messages to other agents. The VDT model computes the frequency of communication among agents working on interdependent tasks based on the task’s interdependence and uncertainty [15]. We have addressed interdependencies, however we require complexity information for each work package in order to model both communication generation and how long an agent, executing an activity, will take to transform a work package.

Thus far, we have defined messages, communication channels, agents, work packages, activities, artifacts, and functions as basic constructs that we need to create a process model and its execution context—i.e. the people, product, and project concerns. When attempting to construct a model for the scenario described in the previous section, it becomes immediately apparent that these constructs are insufficient to describe a model that can capture individual personal processes (for TDD), shared/co-dependent personal processes (for the tester role), work grouping (for grouping work and earning value), the team process (for the overall scrum process), and knowledge (for skills). The remainder of this section focuses on describing how we can apply our constructs and relationships—defining additional ones as required—to model these concerns.

5.1 Modeling Individual Processes

Test driven development (TDD) is a development practice that is summed up as Think, Red, Green, Refactor (Figure 3) [32]. In this process, the developer thinks through the function, develops a test case that fails on execution, develops just enough of the code to ensure the test suite passes, executes the test, then refactors the code ensuring all tests continue to pass. At the end of this process, the feature is complete and the process continues for the next feature.

In this process, the developer works on a single work package, however, there are numerous activities required to complete the work package and thereby complete the function. We define an activity as a construct within our reference model. Activities are partially ordered by dependency and...
may be executed more than once for a given work package.

An interesting implication of this case is that the analysis performed in the refactoring activity may trigger rework in other, completed functions. With TDD, this occurs in the refactor activity. Moreover, the agent may not have sufficient skills to perform the necessary rework. Because we cannot always contain the rework, we need a way to model it in some way. The defect construct supports this, but should rework that does not result in an error impact the quality metric? This decision is left to the modeler; however, we include a rework construct to support either decision, defining a defect as a particular type of rework. We also endow the rework construct with the much of the same attributes as a work package as they both specify work to be performed. Additionally, defects (and potentially rework) may be latent or found. Latent defects are attached to the generated function until found, while found defects are available for fixing.

The TDD process also highlights the fact that activities produce different types of function. For example, the red (test case implementation) activity in TDD generates automated unit tests for the product, but does not add function to the product itself. It would not be correct to attach the test case implementation of the work package to the product artifact, since the test suite may undergo different process rigors as the product artifact. Instead, we wish to add the test code to a separate artifact. We could accomplish this by specifying the target artifact on the work package, but this has the potential to couple the process and product. Instead, we make an indirect reference to an artifact by storing the target artifact type in the activity and, when needed, using the target artifact type to find the particular artifact. The newly generated function object—the automated test cases in our example—is then attached to the found artifact—the test suite.

5.2 Modeling Shared Processes

Some teams split activities among team members. This could be to leverage particular skills/affinities, or, as is often true, reduce confirmation bias in verification and validation activities. Thus, we need to add a mechanism to represent this in our model. Assume we have an activity dependency network as depicted in Figure 4. In this network, no activity may begin until all of its predecessor activities have completed. Assume Understand Work Package Specification must be performed by everyone who works on this work package. Further, assume all activities that do not cross the dotted line must be performed by the same agent or else it must be redone by someone else (as may be the case when there is low ceremony or if new developers are added to the team).

We need a way to model a set of activities that must be performed by the same person. A natural way to represent this is to introduce roles to the model. Roles restrict the type of agent that may perform an activity on a given work package. Further, roles may be reassigned as required, causing loss of effort.

Supplementing the existing model with roles does not prohibit us from allowing an agent to be both a tester and developer. We wish to allow the process to define if this is permissible or not, but specifying roles allows us to define allowable agent-activity assignments in interesting ways—such as an activity that must be performed by each agent working on the work package. We illustrate this with the role annotations in Figure 4.

5.3 Modeling Value Groups

Many agile teams use stories to describe a feature from the user’s perspective [8]. Stories are composed of small tasks, and, upon completion of all tasks, the story is marked as complete. Only when the story is marked as complete may the team claim credit for completing the function. At this point, the product’s functionality is considered to increase—similar to the 0/100 rule in earned value analysis. Thus, in order to model the increase of product function (and value), we require a construct that allows us to group work packages into value groups.

5.4 Modeling Team Processes

Scrum is a popular agile process that allows for a large amount of flexibility. This process consists of five high-level process contexts (Figure 5).

Scrum describes a process in which the team performs work in a fixed-time iteration (sprint) to incrementally develop a product. In this process the development team plans the sprint (the kick-off); executes the plan (the sprint), holding regular stand-up meetings to briefly exchange status; demo the product and gather customer feedback (the product review); and analyze the successes and failures of the previous iteration (the retrospective) [30].

Within each context is a set of child contexts or specific activities. Like activities, contexts at the same level are partially ordered. Each context may specify a set of roles defined for contexts/activities contained within it as well as role assignment constraints. As stated earlier, roles restrict the activities that may be performed by an agent in that role. We expand this so roles can also be used to restrict which contexts may be entered by an agent.

Let’s discuss roles further through an illustrative example.
In the scrum process—its a context—there are four roles defined: the product owner, scrum master, (other) team member, and (other) stakeholder. Let’s say that a team is about to start a scrum meeting—a context within the sprint context. Upon entering the scrum meeting the scrum master, project owner, and team members all behave in the same way, each answering the three status questions. Beyond simply participating as a team member, the scrum master has additional responsibilities within the meeting. Stakeholders, if they are permitted to join the meeting, are generally required to remain quiet. Thus, we see, effectively, three roles within the scrum meeting context: team member, scrum master, and stakeholder. There are a number of ways to model this—including using the parent definitions, overriding the parent roles, or replacing the parent’s roles—each with their own merits. Thus, we want child contexts to support role overriding, accessing parent roles, or removing all parent roles from the agent.

In scrum, there are two databases that help the team track the product’s stories and record plans: the product backlog and sprint backlog [30]. Both decentralize work assignments, empowering developers to retrieve work packages according to pre-established guidelines. It is conceivable that there could be other databases, such as a defect tracking system, to record/store work packages during the simulation. To address this in simulation models, we introduce an additional construct, the backlog, that may contain value groups, work packages, discovered rework, or found defects. In scrum, backlogs may belong to subsets of the team—i.e. certain individuals have responsibilities over the backlog—however, we leave assigning and enforcing such responsibilities as a detail of the input model.

5.5 Modeling Knowledge

Within the sprint, developers work on work packages from the sprint backlog, stopping regularly to participate in the (daily) stand-up meeting. Scrum does not define specific development activities or any form of partial ordering of the activities to be performed when completing a work package. This is left to the team to define should they choose to do so. There are a natural set of activities that developers must perform when developing a function for a given type of artifact as well as a natural partial ordering of those activities.

Further, the scrum process does not define how work packages are assigned to individuals. The team could assign them during the sprint kick-off meeting or individuals could select work packages according to some team-specific high-level principles, such as completing the highest priority work packages first. Work packages may also be assigned according to the combination of resource constraints and qualifications [30].

Resource constraints, or the availability of resources, can be captured as part of each agent. However, when is someone qualified to work on a work package? For software developers and other intellectual workers, fitness for work is based on the individual’s quality of knowledge within a number of different knowledge domains. While there are a number of ways to classify knowledge [11, 29], software development literature generally focuses on declarative knowledge—knowledge based in fact, either semantic knowledge or experiential (episodic) knowledge [29]—and procedural knowledge. Indeed, the process constructs within our reference model encode one form of procedural knowledge. To allow expression of the quality of that procedural knowledge, we wish to include within the agent the ability to deviate from the modeled process and independently sequence activities and contexts.

However, in the authors’ experience, there are a number of knowledge domains that must be modeled including product domain information, product information (e.g. specifications, current progress), and organizational structure—declarative knowledge—as well as skills—procedural knowledge. The reference model, thus far, contains a means to model some product information in the form of work packages and their dependencies. Organizational knowledge should be included within the agent as it describes not just the static structure of the organization, but also the inter-agent relationships and varies by agent. Modeling information and procedures that are required to complete an activity for a work package—including skills, product knowledge, and domain knowledge—requires a new construct to represent this knowledge. We know that an agent requires knowledge to perform work; the quality (or level) of which determines the speed to complete work and quality of the resultant product. However, simply adding knowledge requirements to work packages and activities increases coupling. If, for example, we add skills knowledge to the work package, then we must specify all required skills for any possible activity that could be performed on the work package. Similarly, if we add application/application-domain knowledge or specific technological skills to the activity, we have effectively tied the activity to the project and cannot reuse the construct with different products. In order to separate the product and process, we need to associate knowledge requirements with the work product, activity, and target artifact; we attach domain and application knowledge requirements to the work package, foundational skill requirements (e.g. behavior...
driven development, architecture, or programming skills) to the activity, and specific technology skill requirements (e.g. specific languages or libraries) to the artifact. However, as a reference model, this is simply guidance to those who use this to build a concrete model.

In addition to simply requiring certain pieces of knowledge, the work packages in the IPT model have a difficulty level that must match the knowledge quality (competency-level) of the agent [9]. Using this mechanism, the simulation can ensure the sufficiency of the agent’s knowledge. We incorporate this into the reference model as it will further improve simulation output accuracy, capturing the amount of time spent by the agent acquiring the necessary quality of knowledge. The specifics of agent knowledge acquisition is left to future work.

Attaching knowledge to artifacts poses an additional problem: what happens if there are artifacts that require different skill sets, such as a server written in Java and a client written in .Net? We don’t want to attach these to the same artifact, but in the current model our only choices are to either create two artifacts of the same type, or specify them as two separate types. The former leads to ambiguity for the activity when trying to attach a completed feature to an artifact. The later leads to activity duplication, again tying the process to the product. Rather than introducing ambiguity or greater coupling, we can introduce a new construct, the artifact set, that allows us to group related artifacts. With this construct, the Java server and its related test artifact would be defined in one set, the .Net server and its related test artifact would be in a second set. An integration testing artifact can be defined as needed in a third set or as part of the test artifact of either of the other two sets. To ensure product and process separation, the work package must indicate the artifact set that the activities should target.

### 5.6 Putting it All Together

In developing the reference model, we have explored a number of relationships necessary to model our motivating scenario. For clarity we illustrate many of these constructs and relationships in Figure 6.

Using the constructs defined in our reference model, we show an example of the input model in Listing 1.

Let’s walk through how a simulation of our scenario’s model might progress. For ease of discussion, we assume that the simulation progresses when the simulation clock “ticks”. When the simulation starts, each of the six agents assume a role for the scrum context: the four developers—including aCleverTester—assume the role of team member, the project manager assumes the scrum master role, and the customer representative assumes the product owner role. The scrum project is initialized—causing the product owner to populate the product backlog—and the sprint kickoff meeting is held—populating the sprint backlog. Assume the team communicates with each other and decides that when they enter the Sprint context, they will start with the Develop child-context.

As a team, they enter the sprint context and the team member agents enter the Develop context per the context prerequisites. At scheduled times, the agents suspend existing actions and enter the Stand-up Meeting context. Since there is only one work package that is available for work (i.e. not blocked by prerequisites), two agents work...

<table>
<thead>
<tr>
<th>Agent: aCleverTester</th>
</tr>
</thead>
<tbody>
<tr>
<td>OrganizationStructure: ...</td>
</tr>
<tr>
<td>Knowledge:</td>
</tr>
<tr>
<td>testing, level 5</td>
</tr>
<tr>
<td>java7, level 3</td>
</tr>
<tr>
<td>jUnit4, level 4</td>
</tr>
<tr>
<td>androidSDK, level 2</td>
</tr>
<tr>
<td>androidStudio, level 3</td>
</tr>
<tr>
<td>Product: someApp</td>
</tr>
<tr>
<td>ArtifactSet: myArtifactSet</td>
</tr>
<tr>
<td>Artifact: app</td>
</tr>
<tr>
<td>Type: application</td>
</tr>
<tr>
<td>Knowledge: // Skills</td>
</tr>
<tr>
<td>java7</td>
</tr>
<tr>
<td>androidSDK</td>
</tr>
<tr>
<td>androidStudio</td>
</tr>
<tr>
<td>Artifact: testSuite</td>
</tr>
<tr>
<td>Type: test</td>
</tr>
<tr>
<td>Knowledge: // Skills</td>
</tr>
<tr>
<td>jUnit4</td>
</tr>
<tr>
<td>androidSDK</td>
</tr>
<tr>
<td>androidStudio</td>
</tr>
<tr>
<td>ValueGroup: initialUI</td>
</tr>
<tr>
<td>Value: 5</td>
</tr>
<tr>
<td>WorkPackage: initialView</td>
</tr>
<tr>
<td>ValueGroup: initialUI</td>
</tr>
<tr>
<td>ArtifactSet: myArtifactSet</td>
</tr>
<tr>
<td>Complexity: 5</td>
</tr>
<tr>
<td>Prerequisites: -</td>
</tr>
<tr>
<td>Knowledge: - // Domain Knowledge</td>
</tr>
<tr>
<td>Process: scrum</td>
</tr>
<tr>
<td>Backlog: productBacklog</td>
</tr>
<tr>
<td>Backlog: sprintBacklog</td>
</tr>
<tr>
<td>Context: develop</td>
</tr>
<tr>
<td>Parent: sprint</td>
</tr>
<tr>
<td>Prerequisites: &lt;Agent&gt; Role teamMember</td>
</tr>
<tr>
<td>notEmpty(sprintBacklog)</td>
</tr>
<tr>
<td>Activities:</td>
</tr>
<tr>
<td>implementTests</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>Roles:</td>
</tr>
<tr>
<td>developer</td>
</tr>
<tr>
<td>tester</td>
</tr>
<tr>
<td>Pausable: true</td>
</tr>
<tr>
<td>Properties:</td>
</tr>
<tr>
<td>exitOn(sprintTimeout)</td>
</tr>
<tr>
<td>pauseOn(scrumMeetingTime)</td>
</tr>
<tr>
<td>Activity: implementTests</td>
</tr>
<tr>
<td>Role: Tester</td>
</tr>
<tr>
<td>ArtifactType: test</td>
</tr>
<tr>
<td>Knowledge: // General Skills</td>
</tr>
<tr>
<td>testing</td>
</tr>
<tr>
<td>Prerequisites: exists(&lt;WorkPackage&gt;.ArtifactSet</td>
</tr>
<tr>
<td>ArtifactType: application ,</td>
</tr>
<tr>
<td>Function: &lt;WorkPackage&gt;.name)</td>
</tr>
<tr>
<td>complete(specifyTests)</td>
</tr>
<tr>
<td>Generates: functions</td>
</tr>
</tbody>
</table>

Listing 1: A subset of our scenario’s input expressed using the reference model.
on it (Figure 2). Assume these agents are aCleverTester, who takes on the role of tester, and one of the inexperienced developers takes on the developer role. Each of these agents work on the activities defined for their role (Figure 4), including both of them executing the Understand Work Package Specification activity. Let’s fast forward a bit. Upon completion of an activity, aCleverTester selects an activity to perform by looking at the available activities for its role and determines which one it can perform, based on the prerequisites. aCleverTester notices that all of the prerequisites have been satisfied for the Implement Tests activity, so it performs the Implement Tests activity for the initialView work package. As part of the simulation, aCleverTester determines its outputs for the current tick based upon its knowledge levels compared to the required knowledge levels, the rules in the activity (e.g. find defect/rework, generate latent defect/rework), the work package’s properties, and other properties of the agent. If we assume the simulation progresses in discrete time intervals (ticks), then the output of the agent at the end of a tick could be nothing (i.e. continue to perform the activity for the work package), the newly completed function for the artifact (in this case the TestSuite), or other required objects—messages sent to other agents to improve knowledge, generated rework items, and discovered defects in any of the previously completed function—to complete the work. Generated defects and, potentially, the generated rework are added to the sprint backlog so other agents may work on it.

While working on the project, the agent tracks the amount of time it spent on this project, in terms of ticks, meanwhile gaining development skill and domain knowledge. Upon function completion, the work package is marked complete. The process dictates when to award credit for value group function completion, the work package is marked complete.

In order to simulate requirements changes, we include a special type of agent that generates work package changes (including addition and removal) according to some specified criteria and arrival rate. Change requests, modeled as a type of communication, may include information about a change to an existing work package, changes to the dependency structure, and/or a new work package, depending on the change. It is up to the specific implementation to determine if the generator agent holds onto change requests until queried, if the agent deposits them into a backlog that other agents can pull from, or if the agent notifies others of changes without solicitation. The product owner enacts these changes to the existing work packages in the product backlog if they have not been moved to the sprint backlog, otherwise, it schedules rework to change completed or incomplete work packages that result in the loss of product functionality. Because we are using agents, it is possible to model behaviors in which the product owner agent ignores the prescribed rules and pushes the changes on the team mid-sprint, however defining agent behavior is left to future work.

In a scrum process, the sprint retrospective allows the team to improve the process based on the team’s observation of previous sprints. While this allows for process enhancement in real projects, this is beyond the scope of both our model and planned evaluation environment. However, this meeting does consume time and can have non-tangible effects on the team, thus we still include it in the model and allow the particular agent implementation to model the non-tangible effects, such as morale improvements.

The simulation progresses in much the same way we have described here, and, according to some pre-specified condition, the simulation terminates. The simulator then computes and returns the metrics defined in Section 2.3.
end of the simulation—and dividing it by the scope.

6. DISCUSSION
In this paper we set out to construct an agent-based reference model suitable for describing process simulation models. These models must include constructs that represent the processes as well as the target context—the people, product, and project. At the outset we outlined some requirements for the reference model. The reference model must explicitly support modeling agility and process-product independence.

6.1 Modeling Agility
While our reference model was built around a specific scenario that we wish to model, we believe the constructs and relationships that comprise the reference model are suitable for modeling an arbitrary agile process as the reference model has a means for capturing (1) individual behavior and interactions, (2) in-process requirements changes, (3) trust and low-process ceremony, and (4) incrementally-developed product deliverables.

Our reference model allows implementing models to represent individual behavior through agents as they are suitable abstractions of people [9, 14]. While our reference model does not prescribe the agent’s behavior, it does capture the agent’s relationships to other model constructs. The specific agent representation will dictate the behaviors the agent may or may not express, however the established relationships provide constraints on the otherwise unbounded behavior of the agent.

In our discussion about applying the reference model to our motivational scenario, we described how we could model in-process requirements change through a generator agent. Further, we described how an agent can perform work with low process prescription and ceremony, empowering the agent to perform activities in whichever order it chooses. We also provide a means for agents to communicate or otherwise acquire knowledge when its own knowledge is insufficient to complete an activity for a work package.

Because artifacts are composed of functions in our reference model, we have the means to model incremental product development and feature removal.

6.2 Process Specification Independence
Driven by our desire to provide a means for process experimentation, we set out to decouple the process from the product within the simulation model. With our reference model, we achieve this by breaking direct dependencies among activities and work packages. The work package’s artifact set and the activity’s artifact type properties together provide a means to look up the target artifact of the activity. If the specified artifact exists, it will receive the function once produced; if it doesn’t exist, the activity will be skipped. In this way, we separate the base product and process constructs.

Greater care is required to break coupling caused by knowledge requirements. Earlier, we proposed a division of knowledge requirements that we believe will break the knowledge coupling by associating knowledge with the constructs that need it the most. In our reference model, we propose associating (1) domain and application knowledge requirements with work packages, (2) foundational skills (e.g., testing ability) with activities, and (3) specific technology skills (e.g., languages or libraries) with artifacts. Artifacts, representing the project deliverables, are encoded using specific technologies. In contrast, activities may be used by multiple processes and work packages by multiple activities making both constructs poor candidates for specifying specific technology knowledge requirements. Work packages, describing the properties of the work to be performed, require domain and application knowledge to understand them. Activities, describing how to complete a task in a technology independent way, require generalized knowledge to complete the specified steps. Thus, we have a partitioning of the knowledge in our model that supports process-product independence.

We believe these constructs and relationships sufficiently break the direct dependencies between the specific process and product concerns allowing a modeler to change the details of one without altering the other.

7. NEXT STEPS
The reference model presented here lays the groundwork for holistic process modeling for simulation. Given our particular focus on agile process modeling, we defined our constructs around a specific scenario we wish to model. However, this technique only provides ad hoc validation of the reference model. Other approaches do not validate their base constructs and relationships, rather they evaluate the simulation’s effectiveness. Given enough models, this demonstrates their underlying model’s expressiveness; unfortunately, this fails to measure the quality of their grammar in terms of completeness. We wish to create a grammar from our reference model and systematically evaluate it. Ontological evaluation, using the Bunge-Wand-Weber (BWW) approach [12], shows promise as a method for evaluating both grammar expressiveness and completeness deserving of further investigation.

This work describes the base constructs and the nature of the relationships; however, it does not specify the exact quality of those relationships such as when an activity should complete or when a rework item should be generated. Additional work is required to determine the quality of the relationships described here. [9] and [15] do this for their simulation frameworks by developing a theory to describe their model’s relationships and validating it through comparison to similar theories as well as empirical studies. These studies include surveys to discover the model parameters, case studies to compare simulation outputs to reality, and testing with a human expert oracle to analyze differences in the output from a known good case.

Thus far, we have treated agents as black boxes, describing behavior and externally visible properties in relation to other model constructs. However, agents are a key factor driving the simulation and need to be fully modeled. Minimally, we require a behavior model supporting agent reasoning as well as a learning model. There are a number of different ways this has been done in literature; from agents that generate plans using knowledge collected about the world to agents that select the next action to perform based on their prediction of the utility (or desirability) of the resultant world [31]. While more work is necessary to determine which form of agent will best suit process simulation, we suspect that a utility-driven agent will be a better fit as the agent can easily handle plan deviations.

In this work we constructed an agent-based reference model describing what we believe are the constructs and interactions essential to process modeling for simulation. This model will serve as the basis for a process evaluation envi-
environment that will enable further research into agility and will aid project managers in selecting the right process for their context—i.e. the desired product, the team developing the product, and the project constraints.

8. REFERENCES


