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, Twin-Cities
200 Union St. S.E., Minneapolis, MN 55455, USA

ABSTRACT

Agile development processes are popular when attempting to respond to changing requirements 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. Simulation provides a means to do this. However, in order to simulate agile processes we require both the ability to model individual behavior as well as the decoupling of the process and product. To our knowledge, no existing simulator nor underlying simulation model provide a means to do this. 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 an agile-process evaluation environment.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; I.6.m [**Simulation and Modeling**]: Miscellaneous; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software process*

Keywords

Reference Model, Process Evaluation, Agent-based Simulation, Agile, Process Modeling

1. INTRODUCTION

Agile development processes are intended to allow companies to respond to market change in a controlled manner [1]. 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]. 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 [7, 11], 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.

The paper is structured as follows. In section 2 we lay out the requirements for our reference model—including the desired outputs for a simulation engine implementing the model. We then discuss the current approaches to simulation as they relate to our requirements (section 3). Section 4 describes a small scenario that we will use to construct the reference model in section 5. We discuss the reference model as it relates to our requirements in section 6, and close by describing the next steps in section 7.

ICSSP'15, August 24–26, 2015, Tallinn, Estonia
ACM. 978-1-4503-3346-7/15/08...
http://dx.doi.org/10.1145/2785592.2785615

2. 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.

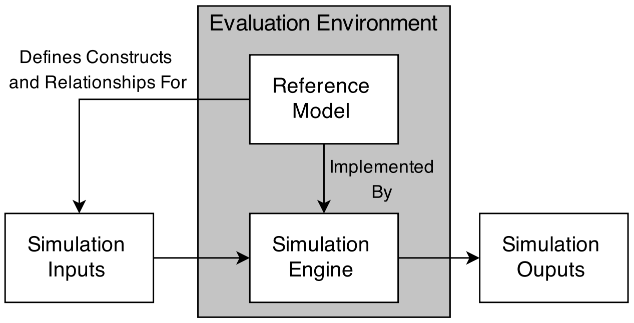

Figure 1: The reference model constructed here defines a grammar for simulation inputs and a high-level design for a concrete simulation engine.

2.1 Modeling Agility

As the mantra goes, "the only thing you can count on is change". This is no less true for software development. Agile processes—as lightweight, iterative and incremental software development processes—provide a means for addressing inevitable requirement changes in a structured manner. These processes emphasize (1) individuals and their interactions; (2) early and frequent customer feedback elicitation; (3) low process ceremony—low control over the team resulting in less documentation and higher levels of informal communication [14]—with a focus on delivering working software [1]. 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 individual team members to perform their job however they see fit. To capture this and 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 stakeholders embedded in the team—i.e. the product owner on some scrum teams [6]. Further, because agile processes emphasize low process ceremony, many agile processes prescribe fewer steps and/or do not define a total ordering of these steps.

Eliciting early and frequent customer feedback is primarily a process concern. However, customer feedback drives requirements change as well as requirement 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 comprehensive documentation" [1]. This necessitates modeling the project deliverables in some way.

2.2 Process Specification Independence

We desire to represent both the product and process within our reference simulation model. Often, models combine software development process activities with the product's work packages into a single concept, called tasks [3, 12, 15, 30]. These tasks are incorporated into a single task network (dependency network) regardless of if they represent an activity or a work package. This coupling has a number of consequences for simulation modeling. First, with tasks capturing both process and product work, process changes can trigger changes in the overall task dependency network, resulting in increased modeling costs and a greater opportunity for model errors. Second, agile processes may not prescribe the exact flow of process activities, leaving activity sequencing to the individual. By coupling process and product work, we inhibit the modeling of individual process activity sequencing and prevent the modeling of individual deviation from the prescribed process. Third, to model changing specifications in a simulation—one of our requirements for modeling agile processes—the simulation engine must generate all tasks that the process prescribes for the work specification and determine where to fit each of them in the project network. This requires detailed process knowledge that cannot simply be encoded within the project network.

We, therefore, desire a simulation model that decouples the process and product work, breaking direct dependencies between them.

2.3 Desired Simulation Outputs

Given that we want to compare processes through simulation, we need to identify the specific metrics we wish to predict in order to create a suitable reference model.

Project success is often measured with relation to project cost, schedule, scope, and quality—i.e. did the project finish on-time, on-budget, and with all features present as specified [2]. Indeed, these constraints—often called the triple constraint[1]—are commonly used to rebalance a project when the project deviates from the plan. However, these concepts are ambiguously defined.

Minimally, we require metrics to cover each of the aspects of the triple constraint. For our schedule metric, we use the duration of the project from the time of the start of the project to its end. Cost is commonly measured in total effort across all individuals, however, since pay is not constant, we wish to include the disaggregated effort—each individual's time expended on the project. Scope metrics are difficult to define because they rely on quantifying the amount of functionality provided by the software product. Often, scope is estimated using source lines of code, which varies significantly due to a number of factors including style, language, and operational definitions [13]. Because point systems, such as function points, provide a consistent mechanism for quanti-

[1]Quality either replaces or is treated as part of the scope depending on the author.

fying functionality without the need for complete knowledge of the target technologies [13], we select these as the basis for the scope metric.

Product quality is particularly difficult to assess. Because processes are viewed as mechanism for ensuring product quality [10,13] and process quality is easier to assess, process quality is often used as an indirect measure of product quality. However, there is an ever-growing body of literature that challenges the credibility of using process quality metrics as indirect product quality measures [22,24]. Rather than predicting quality using 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 [13]. 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 or function points. For consistency, we reuse our previously defined scope metric here.

The metrics defined thus far align with the definitions used in literature [7,19,25]. Although several authors define additional metrics—including the number of generated errors [19] and process productivity [25]—these metrics can be derived from the metrics we defined earlier and could be trivially included in the simulator output 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, Martin and Raffo's hybrid-simulation, is used to evaluate the impact of process changes on project duration, effort, and quality [19]. 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 [16]. 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 [18].

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 [7,30]. 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 discussed earlier, this coupling 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 an important 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.

The virtual design teams (VDTs) simulator [12] 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 actor's current task excluding all previously completed artifacts. In fact, VDT only measures the process quality, not the product quality [12], 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 [17].

Another agent-based simulator, TEAm Knowledge-based Structuring (TEAKS), attempts to support team configuration by predicting team performance [20]. In this simulation, behavior selection is based on fuzzy logic over behavior-influencing characteristics such as personality trends, emotional state, and social traits [20] as well as the assigned task's difficulty and type [21]. This model focuses on the agent interactions rather than accurate estimations [21], and it is unclear what they do to model the agent's environment. Like VDT and IPT, there is no separation of process-from product- tasks. Further, the tasks are fixed on input, precluding the explicit modeling of requirements arrivals.

The Articulator project is an effort to create a metamdoel and agent-based simulation to evaluate processes and predict project outcomes [23]. 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 [23]. 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 [28].

In this section, we have reviewed a number of simulation approaches, none of which satisfy our requirements. The hybrid-simulation and system dynamics models require full prescription of the process activities thereby prohibiting individual process deviation. The articulator meta-model and the models underlying the IPT, VDT, TEAKS simulators fail to separate the product from the process. We, therefore, aim to construct a new reference model that can satisfy our requirements for modeling agility while providing process specification independence.

4. A MOTIVATING SCENARIO

Before we construct our reference model, 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 and ultimately simulate. We will use this scenario to aid us in constructing the reference model.

A team is going to start a mobile app development project and desires to evaluate a set of processes to determine which is going to best fit their needs. The project is time constrained. The product's requirements decomposition is partially depicted in Figure 2. In this illustration, smaller boxes represent work packages—a specification of a small piece of function—and the larger boxes represent non-overlapping groupings of those work packages that will, together, deliver value to the customer upon completion—e.g. stories for most agile teams. In our discussion, work packages do not include process activities.

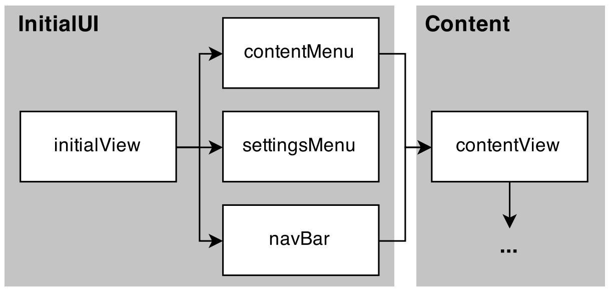

Figure 2: The abbreviated work package dependency network (arrows point to successors).

The team is composed of an inexperienced project manager, a stakeholder that represents the customer, one experienced developer with an affinity for test driven development (TDD), one experienced developer with strong testing skills, and two inexperienced developers.

Having seen scrum [27] work for other teams, the project manager wishes to model and evaluate this process. To reduce resistance to process adoption, he plans to allow individuals to continue to use their preferred personal processes and roles with a common definition of "done". Thus, one developer would act as a tester for others, one would follow TDD, and the remaining developers would determine their own activity sequencing, often using the tester to verify their code.

5. REFERENCE MODEL

Given our previous requirements and the motivating scenario, we construct our reference model from the underlying structures of the scenario we wish to model.

In the reference model's requirements discussion, we described an initial set 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 [7, 11]. 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 [12]. 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.

To aid us in decoupling work packages from process activities, 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 small piece of function within a given artifact. They too are partially ordered according to their dependencies. 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 degree of interdependency and uncertainty [12]. 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) [29]. 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 function is complete and the process continues for the next function.

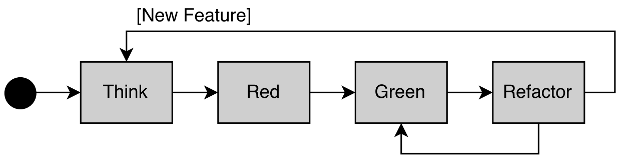

Figure 3: The TDD process [29].

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 already include 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? Rather than limiting concrete model expressiveness, we defer this decision to the modeler and include a rework construct to support either choice; defining a defect as a particular type of rework. Because both work packages and rework specify work to be performed, we endow the rework construct with much of the same attributes as a work package. Additionally, defects (and potentially rework) may be latent or found. Latent defects are attached to the generated function until found. Only 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 indirectly reference the artifact by storing the target artifact type in the activity and, when needed, using it 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 to leverage particular skills/affinities or 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 wish to model a set of activities that must be performed by the same person. A natural way to represent this through roles. Roles restrict the type of agent that may perform an activity on a given work package. Further, roles may be reassigned as required, resulting in lost 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 [4]. 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 feature. At this point, the product's functionality is considered to increase—similar to the 0/100 rule in earned value management [15]. Thus, in order to model the increase of product functionality (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 (Figure 5). In this process, the team performs work in a fixed-time iteration (sprint) to incrementally develop a product. For each sprint, the team plans the sprint (the planning meeting/kick-off); executes the plan, holding regular scrum meetings to briefly exchange status; demo the product and gather customer feedback (the sprint review); and analyze the successes and failures of the previous iteration (the sprint retrospective) [27].

We observe that there are different behaviors exhibited and activities performed within each process context as determined by the developer's role. Further, we observe that each context is composed of a set of child contexts or specific activities. For a developer that uses TDD for his development process, the develop context is composed of the TDD activities: Think, Red, Green, Refactor. The sprint, itself a context, is composed of other contexts—i.e. develop and each of the meetings. 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 there are three roles defined for the team—the product owner, scrum master, and developer—

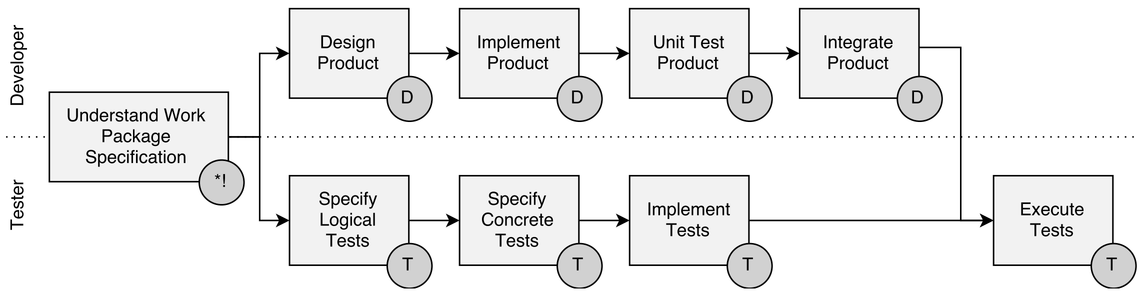

Figure 4: Example activity dependency network (arrows point to successors). The role annotations are shown for a tester ('T') and developer ('D'). '*!' indicates that the activity must be performed by every agent.

with external stakeholders interacting with the team at certain points [27]. Let's say that a team is about to start a scrum meeting. Upon entering the scrum meeting the developers all behave in the same way, each answering the three status questions. This is itself a new behavior for developers specific to the scrum meeting context. The scrum master also has different responsibilities within the meeting context. Stakeholders, including the product owner, are generally required to remain quiet if they are permitted to join the meeting at all. We, therefore, see effectively three roles within the scrum meeting context: developer, scrum master, and stakeholder, each with context-specific actions they may perform. These context-specific actions are different from the behaviors exhibited outside of the scrum meeting context. There are a number of ways to model this behavior change—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 artifacts (databases) that help the team track the product's stories and record plans: the product backlog and sprint backlog [27]. 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 scrum 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

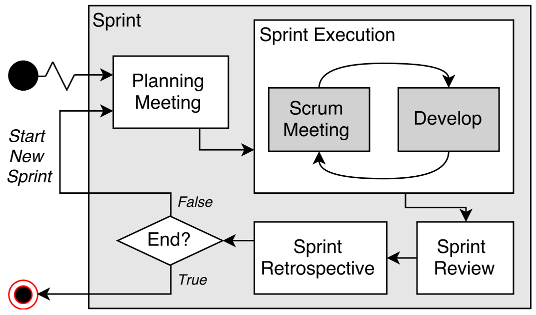

Figure 5: The scrum process. Initial set-up has been omitted.

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 [27].

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 [8, 26], software development literature generally focuses on declarative knowledge—knowledge based in fact, either semantic knowledge or experiential (episodic) knowledge [26]—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 and 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 [7]. 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 function 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

```
Agent: aCleverTester
  OrganizationStructure: ...
  Knowledge:
    testing, level 5
    java7, level 3
    jUnit4, level 4
    androidSDK, level 2
    androidStudio, level 3
    ...

Product: someApp
  ArtifactSet: myArtifactSet
    Artifact: app
      Type: application
      Knowledge: // Skills
        java7
        androidSDK
        androidStudio
    Artifact: testSuite
      Type: test
      Knowledge: // Skills
        java7
        androidSDK
        androidStudio
        jUnit4
  ValueGroup: initialUI
    Value: 5
  WorkPackage: initialView
    ValueGroup: initialUI
    ArtifactSet: myArtifactSet
    Complexity: 3
    Prerequisites: -
    Knowledge: - // Domain Knowledge
    ...

Process: scrum
  Backlog: productBacklog
  Backlog: sprintBacklog
  ...
  Context: develop
    Parent: sprint
    Prerequisites:
      <Agent>.Role == teamMember
      notEmpty(sprintBacklog)
    Activities:
      implementTests
      ...
    Roles:
      developer
      tester
    Pausable: true
    Properties:
      exitOn(sprintTimeout)
      pauseOn(scrumMeetingTime)
  Activity: implementTests
    Role: Tester
    ArtifactType: test
    Knowledge: // General Skills
      testing
    Prerequisites:
      exists(<WorkPackage>.ArtifactSet
              ArtifactType: application,
              Function: <WorkPackage>.name)
      complete(specifyTests)
    Generates: functions
    ...
```

Listing 1: A subset of our scenario's input expressed using the reference model.

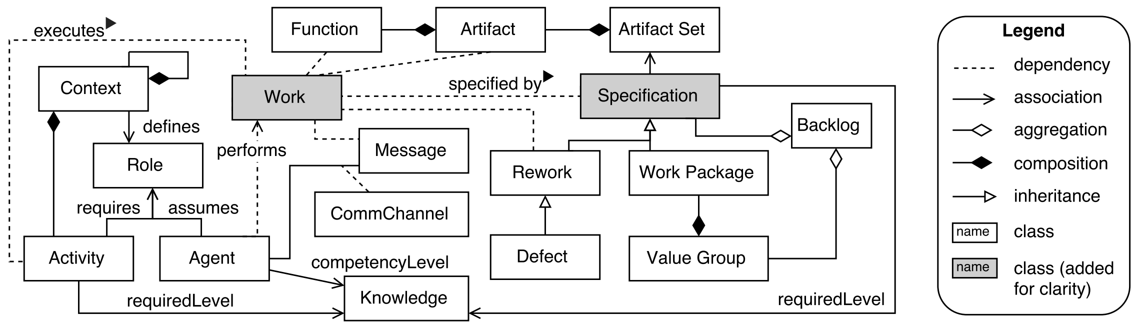

Figure 6: The constructs and relationships in the reference model. The *work* construct above represents the objects required by the agent to complete some work—that is, perform an activity on a work package.

assume a role for the scrum context: the four developers—including `aCleverTester`—assume the developer role, 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 kick-off meeting is held—populating the sprint backlog. Assume the team communicates with each other and decides that when they enter the `Sprint Execution` context, they will start with the `Develop` child-context.

As a team, they enter the sprint context and the developer agents enter the `Develop` context per the context prerequisites. At scheduled times, the agents suspend existing actions and enter the `Scrum Meeting` context.

Since there is only one work package that is available for work (i.e. not blocked by prerequisites), two agents work 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, generate 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, each 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 completion and in our scenario, this is done either during the scrum meeting or upon exiting the `Sprint` context.

Because no other work packages are available and all roles for the available work package have been assigned, the other developer agents idle. This time is not counted toward their time on the project. However, when there is work available for the idling agents and the agents are available, the agents perform work according to their own activity ordering. If the modeler wishes to model preference for particular activity orderings—i.e. strategies—our reference model would support this.

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 in-progress 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 project duration, effort, disaggregated effort, product functionality, and defect density.

Computing the duration and effort metrics is straightforward. As mentioned, our functionality metric increases when all work packages within a value group are completed. As work packages change, we alter the amount of functionality provided by the product, as necessary. There are a number of ways to quantify scope decreases. One such method is to reduce it proportional to the loss of complexity within the value group. We allow this to be defined in the input model. Defect density is computed by summing up the number of unfixed defects—those defects still associated with artifact functions at the 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 using agents as abstractions of people. 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 constrain 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 function 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 [9], 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. A few authors [7, 12] 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 simulation component and must 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 [28]. 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 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. ACKNOWLEDGEMENTS

We would like to thank Gregory Gay, for the insightful reviews of the early reference model.

9. REFERENCES

[1] Manifesto for agile software development. `http://agilemanifesto.org/`, 2001.

[2] Chaos manifesto 2013: Think big, act small. Technical report, The Standish Group International, Inc., 2013.

[3] *A Guide to the Project Management Body of Knowledge (PMBOK® Guide).* Project Management Institute, Incorporated, Newtown Square, PA, 5 edition, 2013.

[4] State of Agile Survey, 9th Annual. `http://www.versionone.com/pdf/state-of-agile-development-survey-ninth.pdf`, 2015.

[5] A. Cockburn. Characterizing people as non-linear, first-order components in software development. Technical Report HaT Technical Report 1999.03, Humans and Technology, Oct. 1999.

[6] M. Cohn. Scrum methodology and project management. `http://mountaingoatsoftware.com/agile/scrum`.

[7] R. Crowder, M. Robinson, H. Hughes, and Y.-W. Sim. The development of an agent-based modeling framework for simulating engineering team work. *IEEE T Syst Man Cy A*, 42(6):1425 –1439, Nov. 2012.

[8] M. Gorman. Types of knowledge and their roles in technology transfer. *J Technol Transfer*, 27(3):219–231, 2002.

[9] P. Green and M. Rosemann. Integrated process modeling: an ontological evaluation. *Inform Syst*, 25(2):73–87, 2000.

[10] W. Humphrey. *Managing the software process.* Addison-Wesley, Reading, MA, 1989.

[11] N. Jennings. On agent-based software engineering. *Artificial intelligence*, 117(2):277–296, 2000.

[12] Y. Jin and R. Levitt. The virtual design team: A computational model of project organizations. *Computational & Mathematical Organization Theory*, 2(3):171–195, Sept. 1996.

[13] S. Kan. *Metrics and Models in Software Quality Engineering.* Addison-Wesley, Toronto, 2 edition, 2003.

[14] K. Knoernschild. Software Process - Maturity, Formality, Ceremony. `http://apsblog.burtongroup.com/2008/11/software-process---maturity-formality-ceremony.html`, Nov. 2008.

[15] E. Larson and C. Gray. *Project Management: The Managerial Process.* McGraw Hill, New York, 5 edition edition, Mar. 2010.

[16] M. Lehman, G. Kahen, and J. Ramil. Simulation process modelling for managing software evolution. In S. Acuña and N. Juristo, editors, *Software Process Modeling*, number 10 in Int Ser Softw Egr, pages 87–109. Springer US, Jan. 2005.

[17] R. Levitt. The Virtual Design Team: Designing Project Organizations as Engineers Design Bridges. *J Organ Des*, 1(2):14, Aug. 2012.

[18] R. Martin and D. Raffo. A model of the software development process using both continuous and discrete models. *Software Process: Improvement and Practice*, 5(2-3):147–157, 2000.

[19] R. Martin and D. Raffo. Application of a hybrid process simulation model to a software development project. *J Syst Software*, 59(3):237–246, 2001.

[20] J. Martínez-Miranda, A. Aldea, R. Bañares-Alcántara, and M. Alvarado. TEAKS: simulation of human performance at work to support team configuration. In *Proc 5th Int Joint Conf on Auton Agent Multi-Ag*, AAMAS '06, pages 114–116, New York, NY, USA, 2006. ACM.

[21] J. Martínez-Miranda and J. Pavón. Modelling Trust into an Agent-Based Simulation Tool to Support the Formation and Configuration of Work Teams. In Y. Demazeau, J. Pavón, J. M. Corchado, and J. Bajo, editors, *7th Int Conf Pract Appl Agents and Multi-Ag*, number 55 in Advances in Intelligent and Soft Computing, pages 80–89. Springer Berlin Heidelberg, 2009.

[22] J. McDermind. Software safety: Where's the Evidence. In *Proc 6th Aust Worksh Saf Crit Sys and Softw*, volume 3 of *SCS '01*. ACM, 2001.

[23] P. Mi and W. Scacchi. A knowledge-based environment for modeling and simulating software engineering processes. *IEEE T Knowl Data En*, 2(3):283 –289, Sept. 1990.

[24] M. Neil and N. Fenton. Predicting software quality using Bayesian belief networks. In *Proceedings of the 21st Annual Software Engineering Workshop*, pages 217–230. NASA Goddard Space Flight Centre, 1996.

[25] L. Putnam and W. Myers. *Five core metrics.* Dorset House Publishing New York, 2003.

[26] P. Robillard. The Role of Knowledge in Software Development. *Commun ACM*, 42(1):87–92, Jan. 1999.

[27] K. Rubin. *Essential Scrum: A practical guide to the most popular Agile process.* Addison-Wesley, 2012.

[28] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River, NJ, 3 edition, 2010.

[29] J. Shore. The Art of Agile Development: Test-Driven Development, Mar. 2010.

[30] Y.-W. Sim, R. Crowder, M. Robinson, and H. Hughes. An agent-based approach to modelling integrated product teams undertaking a design activity. In *Proc ASME Int Des Eng Tech Conf Comput Inf Eng Conf, 2009*, volume 2, pages 227–236, San Diego, CA, Sept. 2009. ASME.